IBM XL C/C++ for Linux, V13.1.6

Compiler Reference
for Little Endian Distributions

Version 13.1.6

IBM XL C/C++ for Linux, V13.1.6

Compiler Reference
for Little Endian Distributions

Version 13.1.6

Contents

About this document

This document is a reference for the IBM® XL C/C++ for Linux, V13.1.6 compiler. Although it provides information about 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, error messages, and return codes.

Who should read this document

This document is for experienced C or C++ developers who have some familiarity with the XL C/C++ compilers or other command-line compilers on Linux 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 about the capabilities and features unique to the XL C/C++ compiler.

How to use this document

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 x.

Throughout this document, the **xlc** and **xlc++** command invocations are used to describe the behavior of the compiler. You can, however, substitute other forms of the compiler invocation command if your particular environment requires it, and compiler option usage remains the same unless otherwise specified.

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

- Compiler installation and system requirements: see the *XL C/C++ Installation Guide* and product README files.
- Overview of XL C/C++ features: see the *Getting Started with XL C/C++*.
- The C or C++ programming language: see the *XL C/C++ Language Reference* for information about the syntax, semantics, and IBM implementation of the C or C++ IBM extension features. See C/C++ standards for the details of standard features.
- Programming topics: see the *XL C/C++ Optimization and Programming Guide* for detailed information about developing applications with XL C/C++, with a focus on program portability and optimization.

How this document 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; and compiler listings and messages.

Chapter 2, "Configuring compiler defaults," on page 17 discusses topics related to setting up default compilation settings, including setting environment variables and customizing the configuration file.

Chapter 3, "Tracking compiler license usage," on page 47 discusses topics related to tracking compiler utilization. This chapter provides information that helps you to detect whether compiler utilization exceeds your floating user license entitlements.

Chapter 4, "Compiler options reference," on page 51 provides a summary of options according to their functional category, through which you can look up and link to options by function. This chapter also includes individual descriptions of selected compiler option sorted alphabetically and a list of the rest of supported GCC options.

Chapter 5, "Compiler pragmas reference," on page 239 provides a list of GCC supported pragmas, which are sorted alphabetically. Then it provides the detailed information of each IBM supported pragma.

Chapter 6, "Compiler commands reference," on page 299 provides detailed descriptions of the commands that are included in XL C/C++.

Chapter 7, "Macros reference," on page 305 provides a list of compiler macros grouped according to their category. It also provides a list of compiler macros that might be supported by other XL compilers but are not supported in IBM XL C/C++ for Linux, V13.1.6.

Chapter 8, "Compiler built-in functions," on page 315 contains individual descriptions of XL C/C++ built-in functions for Power® architectures, categorized by their functionality.

Chapter 9, "OpenMP runtime functions for parallel processing," on page 551 contains individual descriptions of OpenMP runtime library functions for parallel processing.

Conventions

Typographical conventions

The following table shows the typographical conventions used in the IBM XL C/C++ for Linux, V13.1.6 information.

Table 1. Typographical conventions

Typeface	Indicates	Example
bold	Lowercase commands, executable names, compiler options, and directives.	The compiler provides basic invocation commands, **xlc** and **xlC** (**xlc++**), along with several other compiler invocation commands to support various C/C++ language levels and compilation environments.
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.
<u>underlining</u>	The default setting of a parameter of a compiler option or directive.	nomaf | <u>maf</u>

Table 1. Typographical conventions (continued)

Typeface	Indicates	Example
monospace	Programming keywords and library functions, compiler builtins, examples of program code, command strings, or user-defined names.	To compile and optimize myprogram.c, enter: `xlc myprogram.c -03.`

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 begins `C` `C` 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 begins `C++` `C++` 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.
C11 begins `C11` `C11` C11 ends	The text describes a feature that is introduced into standard C as part of C11.
C++11 begins `C++11` `C++11` C++11 ends	The text describes a feature that is introduced into standard C++ as part of C++11.
C++14 begins `C++14` `C++14` C++14 ends	The text describes a feature that is introduced into standard C++ as part of C++14.
IBM extension begins `IBM` `IBM` IBM extension ends	The text describes a feature that is an IBM extension to the standard language specifications.

Table 2. Qualifying elements (continued)

Qualifier/Icon	Meaning
GPU begins ▶ GPU GPU ◀ GPU ends	The text describes the information that is relevant to offloading computations to the NVIDIA GPUs.

Syntax diagrams

Throughout this information, diagrams illustrate XL C/C++ syntax. This section helps 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:

  ►►──keyword──►◄
         └─*optional_argument*─┘

- 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.

  ►►──keyword──┬─*required_argument1*─┬──────────────────────────►◄
           └─*required_argument2*─┘
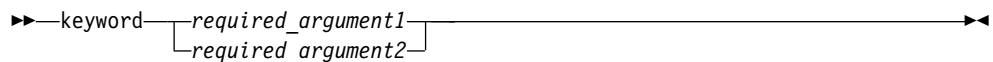
  If choosing one of the items is optional, the entire stack is shown below the main path.

  ►►──keyword──►◄
         ├─*optional_argument1*─┤
         └─*optional_argument2*─┘

- 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:

```
            ┌─ , ◄─────────────────┐
►►──keyword──┴──repeatable_argument─┴──────────────────────────►◄
```

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

```
              ┌─default_argument──┐
►►──keyword───┼───────────────────┼──────────────────────────────►◄
              └─alternate_argument─┘
```
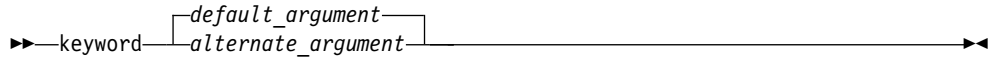
- Keywords are shown in nonitalic letters and should be entered exactly as shown.
- Variables are shown in italicized lowercase letters. They represent user-supplied names or values.
- If punctuation marks, parentheses, arithmetic operators, or other such symbols are shown, you must enter them as part of the syntax.

Example of a syntax statement

EXAMPLE *char_constant* {*a*|*b*}[*c*|*d*]*e*[,*e*]... *name_list*{*name_list*}...

The following list explains the syntax statement:
- Enter the keyword EXAMPLE.
- Enter a value for *char_constant*.
- Enter a value for *a* or *b*, but not for both.
- Optionally, enter a value for *c* or *d*.
- Enter at least one value for *e*. If you enter more than one value, you must put a comma between each.
- Optionally, enter the value of at least one *name* for *name_list*. If you enter more than one value, you must put a comma between each *name*.

Note: The same example is used in both the syntax-statement and syntax-diagram representations.

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 default installation; these need little or no modification.

Related information

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

Available help information
IBM XL C/C++ information

XL C/C++ provides product information in the following formats:
- Quick Start Guide

The Quick Start Guide (`quickstart.pdf`) is intended to get you started with IBM XL C/C++ for Linux, V13.1.6. It is located by default in the XL C/C++ directory and in the \quickstart directory of the installation DVD.

- 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 and subdirectories of the installation DVD.

- 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 Linux, V13.1.6 Installation Guide*.

- Online product documentation

  The fully searchable HTML-based documentation is viewable in IBM Knowledge Center at http://www.ibm.com/support/knowledgecenter/SSXVZZ_13.1.6/com.ibm.compilers.linux.doc/welcome.html.

- PDF documents

  PDF documents are available on the web at https://www.ibm.com/support/knowledgecenter/SSXVZZ_13.1.6/com.ibm.compilers.linux.doc/download_pdf.html.

  The following files comprise the full set of XL C/C++ product information.

  Note: To ensure that you can access cross-reference links to other XL C/C++ PDF documents, download and unzip the .zip file that contains all the product documentation files, or you can download each document into the same directory on your local machine.

Table 3. XL C/C++ PDF files

Document title	PDF file name	Description
What's New for IBM XL C/C++ for Linux, V13.1.6, GC27-8035-00	`whats_new.pdf`	Provides an executive overview of new functions in the IBM XL C/C++ for Linux, V13.1.6 compiler, with new functions categorized according to user benefits.
Getting Started with IBM XL C/C++ for Linux, V13.1.6, GI13-2875-05	`getstart.pdf`	Contains an introduction to XL C/C++, with information about setting up and configuring your environment, compiling and linking programs, and troubleshooting compilation errors.
IBM XL C/C++ for Linux, V13.1.6 Installation Guide, GC27-6540-05	`install.pdf`	Contains information for installing XL C/C++ and configuring your environment for basic compilation and program execution.
IBM XL C/C++ for Linux, V13.1.6 Migration Guide, GC27-8036-00	`migrate.pdf`	Contains migration considerations for using XL C/C++ to compile programs that were previously compiled on different platforms, by previous releases of XL C/C++, or by other compilers.
IBM XL C/C++ for Linux, V13.1.6 Compiler Reference, SC27-6570-05	`compiler.pdf`	Contains information about the various compiler options, pragmas, macros, environment variables, and built-in functions.

Table 3. XL C/C++ PDF files (continued)

Document title	PDF file name	Description
IBM XL C/C++ for Linux, V13.1.6 Language Reference, SC27-6550-05	`langref.pdf`	Contains information about language extensions for portability and conformance to nonproprietary standards.
IBM XL C/C++ for Linux, V13.1.6 Optimization and Programming Guide, SC27-6560-05	`proguide.pdf`	Contains information about advanced programming topics, such as application porting, interlanguage calls with Fortran code, library development, application optimization, and the XL C/C++ high-performance libraries.

To read a PDF file, use Adobe Reader. If you do not have 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, and other articles, is available on the web at http://www.ibm.com/support/docview.wss?uid=swg27036675.

For more information about the compiler, see the XL compiler on Power community at http://ibm.biz/xl-power-compilers.

Other IBM information

- *ESSL product documentation* available at http://www.ibm.com/support/knowledgecenter/SSFHY8/essl_welcome.html?lang=en

Other information

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

Standards and specifications

XL C/C++ is designed to support the following standards and specifications. You can refer to these standards and specifications 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 9899:2011*, also known as *C11*.
- *Information Technology - Programming languages - C++, ISO/IEC 14882:1998*, also known as *C++98*.
- *Information Technology - Programming languages - C++, ISO/IEC 14882:2003*, also known as *C++03*.
- *Information Technology - Programming languages - C++, ISO/IEC 14882:2011*, also known as *C++11*.
- *Information Technology - Programming languages - C++, ISO/IEC 14882:2014*, also known as *C++14* (Partial support).
- *AltiVec Technology Programming Interface Manual*, Motorola Inc. This specification for vector data types, to support vector processing technology, is available at http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf.
- *ANSI/IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-1985.*

- *OpenMP Application Program Interface Version 3.1* (full support), *OpenMP Application Program Interface Version 4.0 (partial support)*, and *OpenMP Application Program Interface Version 4.5 (partial support)*, available at http://www.openmp.org

Other IBM information

- *ESSL product documentation* available at http://www.ibm.com/support/knowledgecenter/SSFHY8/essl_welcome.html?lang=en

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 https://www.ibm.com/support/home/product/Q833644Y89702U61/XL_C/C++_for_Linux. 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 an email to compinfo@cn.ibm.com.

For the latest information about XL C/C++, visit the product information site at https://www.ibm.com/us-en/marketplace/xl-cpp-linux-compiler-power.

How to send your comments

Your feedback is important in helping us 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 to compinfo@cn.ibm.com.

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

Chapter 1. Compiling and linking applications

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

- Preprocessing of program source
- Compiling and assembling into object files
- Linking into an executable

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

The following sections describe how to invoke the XL C/C++ compiler to preprocess, compile, and link source files and libraries:
- "Invoking the compiler"
- "Types of input files" on page 3
- "Types of output files" on page 4
- "Specifying compiler options" on page 5
- "Preprocessing" on page 7
- "Linking" on page 10
- "Compiler messages and listings" on page 11

Invoking the compiler

Different forms of the XL C/C++ compiler invocation commands support various levels of the C and C++ languages. In most cases, you should use the **xlc** command to compile your C source files, and the **xlc++** command to compile C++ source files. Use **xlc++** to link if you have both C and C++ object files.

All the invocation commands can be used to link programs that use multithreading. The **_r** versions of invocation commands are for backward compatibility only.

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/ibm/xlC/13.1.6/etc/xlc.cfg.*$OSRelease.gcc$gccVersion* file for your system.

Examples of the default configuration file are listed below:

- /opt/ibm/xlC/13.1.6/etc/xlc.cfg.sles.12.gcc.4.8.2
- /opt/ibm/xlC/13.1.6/etc/xlc.cfg.rhel.7.3.gcc.4.8.5
- /opt/ibm/xlC/13.1.6/etc/xlc.cfg.centos.7.gcc.4.8.5
- /opt/ibm/xlC/13.1.6/etc/xlc.cfg.ubuntu.16.04.gcc.4.8.2

Table 4. Compiler invocations

Invocations	Description	Equivalent invocations
xlc	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.	xlc_r
c99	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.	c99_r
c89	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.	c89_r
cc	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.	cc_r
xlc++, xlC	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.	xlc++_r, xlC_r

Related information
- "-std (-qlanglvl)" on page 224

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:

```
►►──invocation──┬─input_files──────────────────────┬──────►◄
                └──command_line_options──┘
```
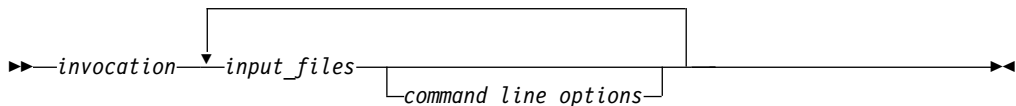
The parameters of the compiler invocation command can be the names of input files, compiler options, and linker options.

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 7 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 **-x 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 **-x c++** option.

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

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 **-x 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 **-x 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.

▶ `GPU` **LLVM IR bitcode libraries**

LLVM IR bitcode libraries must have a .bc suffix. The compiler passes LLVM IR bitcode libraries to the NVVM-IR to PTX translator (llvm2ptx).
`GPU` ◀

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:

"Input control" on page 52

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 **-shared (-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**, 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 **-qmakedep**, **-MD**, or **-MMD** option, a target file suitable for inclusion in a makefile, *file_name*.d is produced for each input file.

Related information:

"Output control" on page 51

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 7.

Related information:

Chapter 4, "Compiler options reference," on page 51

Chapter 5, "Compiler pragmas reference," on page 239

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 7.

Specifying compiler options in a configuration file

The default configuration file (/opt/ibm/xlC/13.1.6/etc/ xlc.cfg.$OSRelease.gcc$gccVersion) defines values and compiler options for the compiler. The compiler refers to this file when compiling C or C++ programs.

Examples of the default configuration file are listed below:
* /opt/ibm/xlC/13.1.6/etc/xlc.cfg.sles.12.gcc.4.8.3
* /opt/ibm/xlC/13.1.6/etc/xlc.cfg.rhel.7.3.gcc.4.8.5
* /opt/ibm/xlC/13.1.6/etc/xlc.cfg.centos.7.gcc.4.8.5
* /opt/ibm/xlC/13.1.6/etc/xlc.cfg.ubuntu.16.04.gcc.4.8.2

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 38.

Specifying compiler options in program source files

You can specify some 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** *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 65, 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 239.

Other pragmas do not have equivalent command-line options; these are described in detail throughout Chapter 5, "Compiler pragmas reference," on page 239.

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, 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. However, some options have cumulative effects when they are specified more than once; examples are the **-I***directory*, **-L***directory*, and **-R***directory_path* options.

When options such as **-qcheck**, **-qfloat**, and **-qstrict** are specified with suboptions for multiple times, each suboption overrides previous specifications of that suboption, but different suboptions are cumulative.

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.

Option	Conflicting options	Resolution
-qfloat=rsqrt	-qnoignerrno	Last option specified
-qfloat=hsflt	-qfloat=spnans	-qfloat=hsflt
-E	-P, -S	-E
-P	-c, -o, -S	-P
-#	-v	-#
-F	-B, -t, -W, -qpath	-B, -t, -W, -qpath
-qpath	-B, -t	-qpath
-S	-c	-S
-nostdinc, -nostdinc++ (-qnostdinc)	-isystem (-qc_stdinc, -qcpp_stdinc, -qgcc_c_stdinc, -qgcc_cpp_stdinc)	-nostdinc, -nostdinc++ (-qnostdinc)

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 75	Preprocesses the source files and writes the output to standard output. By default, `#line` directives are generated.
"-P" on page 84	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.
"-C, -C!" on page 73	Preserves comments in preprocessed output.
"-D" on page 74	Defines a macro name from the command line, as if in a `#define` directive.
-dD[1]	Emits macro definitions to preprocessed output and prints the output.
"-dM (-qshowmacros)" on page 92[1]	Emits macro definitions to preprocessed output.
"-qmakedep, -MD (-qmakedep=gcc)" on page 175	Produces the dependency files that are used by the **make** tool for each source file.
-M[1]	Generates a rule suitable for the **make** tool that describes the dependencies of the input file.
-MD[1]	Compiles the source files, generates the object file, and generates a rule suitable for the **make** tool that describes the dependencies of the input file in a .d file with the name of the input file.
-MF *file*[1]	Specifies the file to write the dependencies to. The **-MF** option must be specified with option **-M** or **-MM**.
-MG[1]	Assumes that missing header files are generated files and adds them to the dependency list without raising an error. The **-MG** option must be used with option **-M**, **-MD**, **-MM**, or **-MMD**.
-MM[1]	Generates a rule suitable for the **make** tool that describes the dependencies of the input file, but does not mention header files that are found in system header directories nor header files that are included from such a header.
-MMD[1]	Compiles the source files, generates the object file, and generates a rule suitable for the **make** tool that describes the dependencies of the input file in a .d file with the name of the input file. However, the dependencies do not include header files that are found in system header directories nor header files that are included from such a header.
-MP[1]	Instructs the C preprocessor to add a phony target for each dependency other than the input file.
-MQ *target*[1]	Changes the target of the rule emitted by dependency generation and quotes any characters that are special to the **make** tool.
-MT *target*[1]	Changes the target of the rule emitted by dependency generation.
"-U" on page 87	Undefines a macro name defined by the compiler or by the **-D** option.

Note:

1. For details about the option, see the GNU Compiler Collection online documentation at http://gcc.gnu.org/onlinedocs/.

Directory search sequence for included files

The XL C/C++ compiler supports the following types of included 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 **-include** 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 **-include** only: The compiler searches the current (working) directory from which the compiler is invoked.[1]
2. Header files included with **-include** or `#include "file_name"`: The compiler searches the directory in which the source file is located.
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: The compiler searches the standard directory for the system headers. The default directory 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 **-isystem** (**-qgcc_c_stdinc** or **-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 **-nostdinc** or **-nostdinc++** (**-qnostdinc**) compiler option is in effect, step 4 is omitted.

Related information

- "-I" on page 78
- "-isystem (-qc_stdinc) (C only)" on page 123
- "-isystem (-qcpp_stdinc) (C++ only)" on page 124
- "-isystem (-qgcc_c_stdinc) (C only)" on page 125
- "-isystem (-qgcc_cpp_stdinc) (C++ only)" on page 127
- "-qidirfirst" on page 155

- "-include (-qinclude)" on page 122
- "-qstdinc, -qnostdinc (-nostdinc, -nostdinc++)" on page 206

Linking

The linker links specified object files to create one executable file. Invoking the compiler with one of the invocation commands automatically calls the linker unless you specify one of the following compiler options:

- **-c**
- **-E**
- **-M**
- **-P**
- **-S**
- **-fsyntax-only (-qsyntaxonly)**
- **-### (-#)**
- **--help (-qhelp)**
- **--version (-qversion)**

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 a .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:

xlc myfile.c -o myfile

If you use the **-shared (-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 63.

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

- Use **-t** and **-B** or use **-qpath** to specify the nondefault linker, for example,

-tl -B*linker_path*

or

-qpath=l:*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
- "-shared (-qmkshrobj)" on page 220

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 63
- "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 your application have the packages that contain the libraries. To make sure the
required libraries are available to the users of your application, take one of the
following actions:

- Ship the packages that contain the redistributable libraries with your application.
  The packages are stored under the `images/rpms` directory in the installed
  compiler package..

- Direct the users of your application to download the appropriate runtime
  libraries from the *Latest updates for supported IBM C and C++ compilers* link from
  the XL C/C++ support website at https://www.ibm.com/support/home/
  product/Q833644Y89702U61/XL_C/C++_for_Linux.

For information about the licensing requirements related to the distribution of
these packages, see the LicenseAgreement.pdf file in the installed compiler
package.

Table 5. Redistributable libraries

Package name	Libraries (and default installation path)	Description
libxlc-devel	/opt/ibm/xlC/13.1.6/lib/libxl.a /opt/ibm/xlC/13.1.6/lib/libxlopt.a	XL C/C++ compiler libraries
vacpp.rte	/opt/ibmcmp/vac/13.1.6/lib/libibmc++.so.1	XL C++ runtime libraries

Compiler messages and listings

The following sections discuss the various information generated by the compiler
after compilation.

- "Compiler messages" on page 12
- "Compiler listings" on page 13
- "Paging space errors during compilation" on page 15

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. You can control which code constructs cause the compiler to emit errors and warning messages and how they are displayed to the console.

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 table below provides a key to the abbreviations for the severity levels and the associated default compiler response.

You can use the **-Werror (-qhalt=w)** option to stop the compilation for warnings and all types of errors.

You can use the **-Werror=unused-command-line-argument** option to switch between warnings and errors for invalid options.

Table 6. Compiler message severity levels

Letter	Severity	Synonym	Compiler response
I	Informational	note	Compilation continues and object code is generated. The message reports conditions found during compilation.
W	Warning	warning	Compilation continues and object code is generated. The message reports valid but possibly unintended conditions.
▶ C ◀ E	Error	error	Compilation continues and object code is generated. The compiler can correct the error conditions that are found, but the program might not produce the expected results.
S	Severe error	error	Compilation continues, but object code is not generated. The compiler cannot correct the error conditions that are found. • 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 those options. • Check for and correct any other errors reported prior to the severe error. • If the message indicates an internal compile-time error, report the message to your IBM service representative.
▶ C ◀ U	Unrecoverable error	fatal error	The compiler halts. An internal compile-time error has occurred. Report the message to your IBM service representative.

Related information
- "-Werror (-qhalt)" on page 90
- "Listings, messages, and compiler information" on page 60

Compiler return codes

At the end of compilation, the compiler sets the return code to zero when no messages are 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.

gxlc and gxlc++ return codes

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

40   A gxlc or gxlc++ option error or unrecoverable error has been detected.

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

Compiler listings

A listing is a compiler output file (with a .lst suffix) that contains information about a particular compilation. As a debugging aid, a compiler listing is useful for determining what has gone wrong in a compilation.

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

- -qlist
- -qreport

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.

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.

PDF report section

The following information is included in this section when you use the **-qreport** option with the **-qpdf2** option:

Loop iteration count
> The most frequent loop iteration count and the average iteration count, for a given set of input data, are calculated for most loops in a program. This information is only available when the program is compiled at optimization level **-O5**.

Block and call count
> This section covers the *Call Structure* of the program and the respective execution count for each called function. It also includes *Block information* for each function. For non-user defined functions, only execution count is given. The Total Block and Call Coverage, and a list of the user functions ordered by decreasing execution count are printed in the end of this report section. In addition, the Block count information is printed at the beginning of each block of the pseudo-code in the listing files.

Cache miss
> This section is printed in a single table. It reports the number of *Cache Misses* for certain functions, with additional information about the functions such as: `Cache Level`, `Cache Miss Ratio`, `Line Number`, `File Name`, and `Memory Reference`.
>
> **Note:** You must use the option **-qpdf1=level=2** to get this report. You can also select the level of cache to profile using the environment variable **PDF_PM_EVENT** during run time.

Relevance of profiling data
> This section shows the relevance of the profiling data to the source code during the PDF1 step. The relevance is indicated by a number in the range of 0 - 100. The larger the number is, the more relevant the profiling data is to the source code, and the more performance gain can be achieved by using the profiling data.

Missing profiling data
> This section might include a warning message about missing profiling data. The warning message is issued for each function for which the compiler does not find profiling data.

Outdated profiling data
> This section might include a warning message about outdated profiling data. The compiler issues this warning message for each function that is modified after the PDF1 step. The warning message is also issued when the optimization level changes from the PDF1 step to the PDF2 step.

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

Object section

If you specify the **-qlist** option, the Object section 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
- "Listings, messages, and compiler information" on page 60

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, take any of the following actions 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.

Running user-defined actions by using Clang plug-ins

You can write Clang plug-ins to run extra user-defined actions during compilation.

Note: To use Clang plug-ins, you must build your plug-ins by using Clang 4.0.

Procedure

1. Write your plug-in. For more information about writing Clang plug-ins, see the Clang 4.0 documentation at http://www.llvm.org/releases/4.0.0/tools/clang/docs/ClangPlugins.html.
2. Register the plug-in in a dynamic library.
3. Run the plug-in during compilation.

   Note: You must prefix each Clang plug-in option and argument with **-Xclang**.

   a. Specify the **-load** option to load the dynamic library in which the plug-in is registered. All plug-ins that are registered in that dynamic library are loaded. You can specify this option multiple times to load several libraries for dispersed plug-ins.

b. Specify the **-add-plugin** option to select the plug-in to run. You can specify this option multiple times to run several plug-ins.

c. Optional: Specify arguments for the plug-in by using the **-plugin-arg-**_plugin_name_ option.

Example

In the following example, `CountVariableUsage.so` is the dynamic library that contains the target plug-in `count-vars`, and `-static-only` is the argument for the `count-vars` plug-in.

```
xlC myprogram.cpp –Xclang –load –Xclang CountVariableUsage.so \
-Xclang –add-plugin –Xclang count-vars \
-Xclang -plugin-arg-count-vars -Xclang –static-only
```

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.
- Settings defined in the compiler configuration file, `xlc.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 38.

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 18
- "Runtime environment variables" on page 18

Compile-time and link-time environment variables

The following environment variables are used by the compiler when you are compiling and linking your code. Many are built into the Linux operating system. With the exception of *LANG*, 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 85.

NLSPATH

Specifies the directory search path for finding the compiler message and help files. 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/ibm/xlC/13.1.6/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 38.

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.

PDF_BIND_PROCESSOR

If you want to bind your process to a particular processor, you can specify the PDF_BIND_PROCESSOR environment variable to bind the process tree from the executable to a different processor. Processor 0 is set by default.

PDF_PM_EVENT

When you run an application compiled with **-qpdf1=level=2** and want to gather different levels of cache-miss profiling information, set the PDF_PM_EVENT environment variable to L1MISS, L2MISS, or L3MISS (if applicable) accordingly.

PDF_SIGNAL_TO_DUMP

If you want to dump snapshot PDF profiling information to files during execution, you must define the PDF_SIGNAL_TO_DUMP environment variable before running the application. The value must be an integer in the range of SIGRTMIN and SIGRTMAX inclusive. For more information, see "Dumping snapshot PDF profiling information to files during execution"in the *XL C/C++ Optimization and Programming Guide*.

PDF_WL_ID

This environment variable is used to distinguish the sets of PDF counters that are generated by multiple training runs of the user program. Each run receives distinct input.

By default, PDF counters for training runs after the first training run are added to the first and the only set of PDF counters. This behavior can be changed by setting the PDF_WL_ID environment variable before each PDF training run. You can set PDF_WL_ID to an integer value in the range 1 - 65535. The PDF runtime library then uses this number to tag the set of PDF counters that are generated by this training run. After all the training runs complete, the PDF profile file contains multiple sets of PDF counters, each set with an ID number.

PDFDIR

Optionally specifies the directory in which profiling information is saved when you run an application that you have compiled with the **-qpdf1** option. The default value is unset, and the compiler places the profile data file in the current working directory. If the PDFDIR environment variable is set but the specified directory does not exist, the compiler issues a warning message. When you recompile or relink your program with the **-qpdf2** option, the compiler uses the data saved in this directory to optimize the application. It is recommended that you set this variable to an absolute path if you use profile-directed feedback (PDF). See "-qpdf1, -qpdf2" on page 180 for more information.

Environment variables for parallel processing

The XLSMPOPTS environment variable sets options for program run time using loop parallelization. For more information about the suboptions for the **XLSMPOPTS** environment variables, see "XLSMPOPTS" on page 20.

If you are using OpenMP constructs for parallelization, you can also specify runtime options using the OMP environment variables, as discussed in "Environment variables for OpenMP" on page 25.

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

Related information
• "Pragma directives for OpenMP parallelization" on page 253

XLSMPOPTS

You can specify runtime options that affect parallel processing by using the **XLSMPOPTS** environment variable. This environment variable must be set before you run an application. The syntax is as follows:

```
>>--XLSMPOPTS-- =---+-----+--+--runtime_option_name-- =----option_setting----+--+-----+--><
                    '- " -'  |  '-------------------- : <---------------------'  '- " -'
```
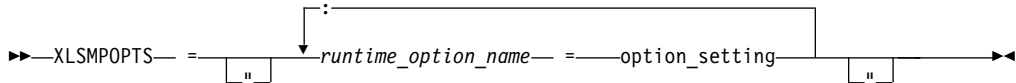
You can specify option names and settings in uppercase or lowercase. You can add blanks before and after the colons and equal signs to improve readability. However, if the **XLSMPOPTS** option string contains imbedded blanks, you must enclose the entire option string in double quotation marks (").

For example, to have a program run time create 4 threads and use dynamic scheduling with chunk size of 5, you can 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"
• "Parallel environment options" on page 21
• "Offloading computations to target devices options" on page 23
• "Performance tuning options " on page 23
• "Dynamic profiling options" on page 24

Scheduling options

schedule

Specifies the type of scheduling algorithms and chunk size (n) that are used for automatic parallelization on loops to which no other scheduling algorithm has been explicitly assigned in the source code. Automatic parallelization is enabled by the **-qsmp=auto** option.

Note: Use the **OMP_SCHEDULE** environment variable for loops that are explicitly assigned to runtime schedule type with the **OpenMP** schedule clause.

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 is not part of the OpenMP API standard.

Note: This suboption has been deprecated and might be removed in a future release. Instead, you can use the **guided** suboption.

dynamic[=*n*]
The iterations of a loop are divided into chunks that contain *n* contiguous iterations each. The final chunk might contain fewer than *n* iterations. If *n* is not specified, the default chunk size is one.

Each thread is initially assigned one chunk. After threads complete their assigned chunks, they are assigned remaining chunks on a "first-come, first-do" basis.

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. The final chunk might contain fewer than n 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 **floor**(*number_of_iterations*/ *number_of_threads*) iterations. The first **remainder**(*number_of_iterations*/ *number_of_threads*) chunks have one more iteration. 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.

If you specify **schedule** with no suboption, the scheduling type is determined at run time.

Parallel environment options

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.

Note: This option has been deprecated and might be removed in a future release.

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.

Note: This option has been deprecated and might be removed in a future release.

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.

Note: This option has been deprecated and might be removed in a future release. Instead, you can use the **OMP_STACKSIZE** environment variable.

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, and it must be a nonzero positive number. 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. For more information, see "-qsmp" on page 201.

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.

Note: This option has been deprecated and might be removed in a future release. Instead, you can use the **OMP_PLACES** environment variable.

procs=*cpu_id[,cpu_id,...]*
: Enables thread binding and specifies a list of *cpu_id* to which the threads are bound.

Note: This option has been deprecated and might be removed in a future release. Instead, you can use the **OMP_PLACES** environment variable.

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.

Note: This option has been deprecated and might be removed in a future release. Instead, you can use the **OMP_PLACES** environment variable.

Offloading computations to target devices options

▶ GPU

target
>   Controls which device to execute target regions on. The syntax for this option is **target**=*suboption*, where the suboptions are defined as follows:
>
>   **mandatory**
>   >   Forces target regions directed at a target device to execute on the target device. Program aborts if target regions cannot execute on the target device. This is the default value for machines with devices.
>
>   **default**
>   >   Executes target regions directed at a target device on the target device when the device and binaries are available. If a target region fails in executing on the target device, all subsequent target regions will execute on the host device.
>
>   **disabled**
>   >   Executes all target regions on the host device. This is the default value for machines without devices.

cudamemcheckfriendly
>   Controls whether to disable the check for pinned memory in the runtime. The syntax for this option is **cudamemcheckfriendly**=*suboption*, where the suboptions are defined as follows:
>
>   **off**
>   >   Enables the check for pinned memory in the runtime. This is the default value.
>
>   **on**   Disables the check for pinned memory in the runtime. Specify this suboption if you want to run the cuda-memcheck tool from the NVIDIA CUDA Toolkit on your application.

GPU ◀

Performance tuning options

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

profilefreq=*num*
> 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, which are described below.

> The valid 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.

> **Note:** 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**.

Related reference:

"OMP_STACKSIZE" on page 36

-qsmp

Related information:

"OMP_PLACES" on page 30

Environment variables for OpenMP

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

```
►►──env_variable──=──option_and_args──────────────────────────────────►◄
```

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

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

OMP_DEFAULT_DEVICE: The **OMP_DEFAULT_DEVICE** environment variable controls the device number to be used in device constructs by setting the initial value of the *default-device-var* internal control variable (ICV).

```
►►──OMP_DEFAULT_DEVICE=──n─────────────────────────────────────────────►◄
```

n   Is a non-negative integer value that is less than the value of the **omp_get_num_devices** function.

Related reference:

"omp_get_num_devices" on page 553

OMP_DISPLAY_ENV: When a program that uses the OpenMP runtime is invoked and the **OMP_DISPLAY_ENV** environment variable is set, the OpenMP runtime library displays the values of the internal control variables (ICVs) associated with the environment variables and the build-specific information about the runtime library.

OMP_DISPLAY_ENV is useful in the following cases:
- When the runtime library is statically linked with an OpenMP program, you can use **OMP_DISPLAY_ENV** to check the version of the library that is used during link time.
- When the runtime library is dynamically linked with an OpenMP program, you can use **OMP_DISPLAY_ENV** to check the library that is used at run time.
- You can use **OMP_DISPLAY_ENV** to check the current setting of the runtime environment.

By default, no information is displayed.

The syntax of this environment variable is as follows:

```
►►──OMP_DISPLAY_ENV──=──┬──TRUE────┬──────────────────────────────────►◄
                        ├──FALSE───┤
                        └──VERBOSE─┘
```

Note: The values **TRUE**, **FALSE**, and **VERBOSE** are not case-sensitive.

TRUE
Displays the OpenMP version number defined by the _OPENMP macro and the initial ICV values for the OpenMP environment variables.

FALSE
Instructs the runtime environment not to display any information.

VERBOSE

Displays build-specific information, ICV values associated with OpenMP environment variables, and the setting of the **XLSMPOPTS** environment variable.

Usage

When **OMP_DISPLAY_ENV** is **TRUE**, the initial ICV values for the OpenMP environment variables are displayed. If **OMP_PLACES** is set to **cores** or **threads**, the **OMP_PLACES** value is displayed in the format of **cores** or **threads** followed by the number of places in brackets; for example, **OMP_PLACES='cores(4)'**. For custom **OMP_PLACES**, each resource is displayed individually in each place, followed by the keyword **custom**; for example, **OMP_PLACES='{4,5,6,7},{8,9,10,11}' custom**.

When **OMP_DISPLAY_ENV** is **VERBOSE**, the output includes a section that is delineated by the lines OPENMP DISPLAY AFFINITY BEGIN and OPENMP DISPLAY AFFINITY END. This section includes a verbose display of the **OMP_PLACES** value, where each resource for each place is displayed individually, followed by **cores**, **threads**, or **custom** as appropriate. This section also displays information on **THREADS_PER_PLACE** in the format of a comma-separated list of the individual **THREADS_PER_PLACE** value for each place; for example, **THREADS_PER_PLACE='{2},{2}'**.

Examples

Example 1

If you enter the **export OMP_DISPLAY_ENV=TRUE** command, you will get output that is similar to the following example:

```
OPENMP DISPLAY ENVIRONMENT BEGIN
  OMP_DISPLAY_ENV='TRUE'

  _OPENMP='201107'
  OMP_DYNAMIC='FALSE'
  OMP_MAX_ACTIVE_LEVELS='5'
  OMP_NESTED='FALSE'
  OMP_NUM_THREADS='96'
  OMP_PROC_BIND='FALSE'
  OMP_SCHEDULE='STATIC,0'
  OMP_STACKSIZE='4194304'
  OMP_THREAD_LIMIT='96'
  OMP_WAIT_POLICY='PASSIVE'
OPENMP DISPLAY ENVIRONMENT END
```

Example 2

If you enter the **export OMP_DISPLAY_ENV=VERBOSE** command, you will get output that is similar to the following example:

```
OPENMP DISPLAY AFFINITY BEGIN
 OMP_PLACES='{0},{1},{2},{3},{4},{5},{6},{7},{8},{9},{10}' cores
 THREADS_PER_PLACE='{1},{1},{1},{1},{1},{1},{1},{1},{1},{1},{1}'
OPENMP DISPLAY AFFINITY END
```

Related information:

"XLSMPOPTS" on page 20

"OMP_PLACES" on page 30

"OMP_PROC_BIND" on page 32

OMP_DYNAMIC: The **OMP_DYNAMIC** environment variable controls dynamic adjustment of the number of threads available for running parallel regions.

```
              ┌─TRUE──┐
►►──OMP_DYNAMIC──=──┤       ├────────────────────────────────────────►◄
              └─FALSE─┘
```
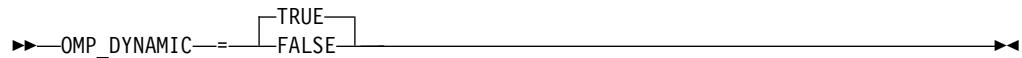
When **OMP_DYNAMIC** is set to TRUE, the number of threads that are created and then assigned to a place must not exceed the value of **THREADS_PER_PLACE**. The thread number includes the currently allocated threads of all active parallel regions. Under a given **OMP_PROC_BIND** policy, **THREADS_PER_PLACE** takes precedence in all situations.

When **OMP_DYNAMIC** is set to FALSE, if an application requires more threads than the value of **THREADS_PER_PLACE** in any place under a given **OMP_PROC_BIND** policy, the excess threads beyond the value of **THREADS_PER_PLACE** for all such places are assigned with priority to the following places:

1. Places that have not reached **THREADS_PER_PLACE**.
2. Places where the master thread is not running.

Examples

Example 1

Suppose **OMP_THREAD_LIMIT**=48 and **OMP_PLACES**={0,1,2,3,4,5,6,7},{8,9,10,11,12,13,14,15},{16,17,18,19}, the **THREADS_PER_PLACE** values are calculated as follows:

P0={0,1,2,3,4,5,6,7}: size = 8, total size = 20, **THREADS_PER_PLACE** = floor((8/20)*48) = floor(19.2) = 19

P1={8,9,10,11,12,13,14,15}: size = 8, total size = 20, **THREADS_PER_PLACE** = floor((8/20)*48) = floor(19.2) = 19

P2={16,17,18,19}: size = 4, total size = 20, **THREADS_PER_PLACE** = floor((4/20)*48) = floor(9.6) = 9

The number of total allocated threads is 47. Threads are allocated by place size. Because P0 and P1 have the same largest size and P0 comes first in **OMP_PLACES**, threads are allocated starting with P0. The thread allocation order is: P0, P1, P2. Only one thread is unallocated, so it is allocated to P0. Therefore, **THREADS_PER_PLACE**={20},{19},{9}.

Example 2

Suppose **OMP_THREAD_LIMIT**=17 and **OMP_PLACES**={0,1,2,3,0,1,2,3},{4,5,6,7,},{8,9,10,11}, the **THREADS_PER_PLACE** values are calculated as follows:

P0={0,1,2,3,0,1,2,3}: size = 8, total size = 16, **THREADS_PER_PLACE** = floor((8/16)*17) = floor(8.5) = 8

P1={4,5,6,7}: size = 4, total size = 16, **THREADS_PER_PLACE** = floor((4/16)*17) = floor(4.25) = 4

P2={8,9,10,11}: size = 4, total size = 16, **THREADS_PER_PLACE** = floor((4/16)*17) = floor(4.25) = 4

The number of total allocated threads is 16. Threads are allocated by place size, so the thread allocation order is: P0, P1, P2. Only one thread is unallocated, so it is allocated to P0. Therefore, **THREADS_PER_PLACE**={9},{4},{4}.

Example 3

Suppose **OMP_THREAD_LIMIT**=394 and **OMP_PLACES**={0,1},{2,3,4,5}, {6,7,8,9,10,11},{12,13,14,15},{16,17,18,19,20,21,22,23}, the **THREADS_PER_PLACE** values are calculated as follows:

P0={0,1}: size = 2, total size = 24, **THREADS_PER_PLACE** = floor((2/24)*394) = floor(32.8) = 32

P1={2,3,4,5}: size = 4, total size = 24, **THREADS_PER_PLACE** = floor((4/24)*394) = floor(65.7) = 65

P2={6,7,8,9,10,11}: size = 6, total size = 24, **THREADS_PER_PLACE** = floor((6/24)*394) = floor(98.5) = 98

P3={12,13,14,15}: size = 4, total size = 24, **THREADS_PER_PLACE** = floor((4/24)*394) = floor(65.7) = 65

P4={16,17,18,19,20,21,22,23}: size = 8, total size = 24, **THREADS_PER_PLACE** = floor((8/24)*394) = floor(131.3) = 131

The number of total allocated threads is 391. Threads are allocated by place size, so the thread allocation order is: P4, P2, P1, P3, P0. Three threads are unallocated, so the **THREADS_PER_PLACE** values of P4, P2, and P1 are increased by one each. Therefore, **THREADS_PER_PLACE**={32},{66},{99},{65},{132}.

Related information

"OMP_PLACES" on page 30

"OMP_PROC_BIND" on page 32

OMP_MAX_ACTIVE_LEVELS:
The **OMP_MAX_ACTIVE_LEVELS** environment variable sets the *max-active-levels-var* internal control variable. This controls the maximum number of active nested parallel regions.

```
►►──OMP_MAX_ACTIVE_LEVELS=n───────────────────────────────────────►◄
```

n   is the maximum number of nested active parallel regions. It must be a positive scalar integer. The maximum value that you can specify is 5.

In programs where nested parallelism is enabled, the initial value is greater than 1. The function **omp_get_max_active_levels** can be used to retrieve the *max-active-levels-var* internal control variable at run time.

OMP_NESTED: The **OMP_NESTED** environment variable enables or disables nested parallelism. The syntax is as follows:

```
                          ┌FALSE┐
►►──OMP_NESTED=──┴─TRUE─┴────────────────────────────────────────────►◄
```

If you set this environment variable to **TRUE**, nested parallelism is enabled, which means that the runtime environment might deploy extra threads to form the team of threads for the nested parallel region. If you set this environment variable to **FALSE**, nested parallelism is disabled, which means nested parallel regions are serialized and run in the encountering thread.

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

The setting of the **omp_set_nested** function overrides the **OMP_NESTED** setting.

Note: If the number of threads in a parallel region and its nested parallel regions exceeds the number of available processors, your program might suffer performance degradation.

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:

```
►►──OMP_NUM_THREADS=──num_list─────────────────────────────────────►◄
```

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. If nested parallelism is disabled, 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**), and 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.

   Note: The **parthds** suboption of the **XLSMPOPTS** environment variable is deprecated.

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_PLACES: The **OMP_PLACES** environment variable specifies a list of places that are available when the OpenMP program is executed. The value of **OMP_PLACES** can be either one of the following values:
- An explicit list of places that are described by non-negative numbers
- An abstract name that describes a set of places

OMP_PLACES syntax

```
►►──OMP_PLACES=──┬──place_list──┬───────────────────────────────────────────────►◄
                 └──place_name──┘
```

where *place_list* takes one of the following syntax forms:

place_list syntax: form 1

```
              ┌──,────────────────────────────────────────────┐
              │                                                │
►►────────────┴──┬──!──┬──{──┬──lower_bound──:──length──────┬──}─┴───────────►◄
                 └─────┘     │             └──:──stride──┘   │
                             │  ┌──,──┐                      │
                             └──┴──num─┴──────────────────────┘
```

place_list syntax: form 2

```
       ┌──!──┐
►►─────┴─────┴──{──lower_bound──:──length──}──:──num_places──:──multiplier──────►◄
```

where *lower_bound*, *length*, *stride*, *num*, *num_places*, and *multiplier* are positive
integers. The thread number in each place starts with the value that is a multiple
of *multiplier*. The exclusion operator **!** excludes the number or place that follows
the operator immediately.

place_name syntax

```
        ┌──cores───┐
►►──────┼──threads──┼──────────────────────────────────────────────────────────►◄
        └──────────┘──└──(──num_places──)──┘
```

threads
> Each place contains a hardware thread.

cores
> Each place contains a core. If **OMP_PLACES** is not set, the default setting is
> **cores**.

num_places
> Is the number of places.

Usage

When requested places are fewer than that are available on the system, the
execution environment assigns places in the order of the place list at run time.
When requested places are more than that are available on the system, the
execution environment assigns the maximum number of places that the system
supports at run time.

For a program that contains both OpenMP and OpenMPI code, the OpenMP
runtime detects the existence of OpenMPI code by the presence of the
OMPI_COMM_WORLD_RANK environment variable. If you do not set **OMP_PLACES**
explicitly, the compiler sets **OMP_PLACES** to **cores** and removes any unavailable
resources from **OMP_PLACES** based on the OpenMPI affinity policy. In addition,
OMP_PROC_BIND is set to **TRUE**.

For examples on how to set the **OMP_PLACES** environment variable, see examples in OMP_PROC_BIND.

OMP_PROC_BIND: The **OMP_PROC_BIND** environment variable controls the thread affinity policy and whether OpenMP threads can be moved between places. With the thread affinity feature, you can have a fine-grained control of how threads are bound and distributed to places. The thread affinity policies are **MASTER**, **CLOSE**, and **SPREAD**.

OMP_PROC_BIND syntax

```
►►──OMP_PROC_BIND=──┬─TRUE──────────────────────────────────────────────►◄
                    ├─FALSE─────────────────┤
                    │  ┌─,────────────┐     │
                    │  ▼              │      │
                    └─────┬─MASTER─┬──┘
                          ├─CLOSE──┤
                          └─SPREAD─┘
```
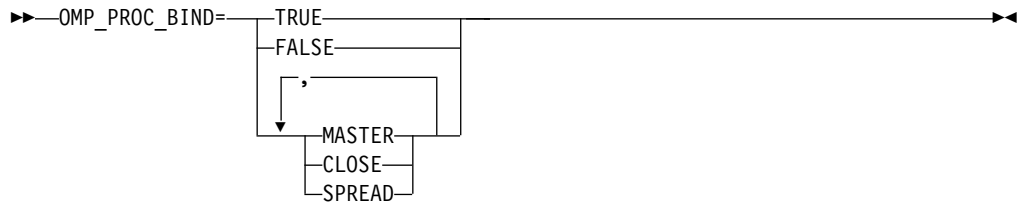
TRUE
> Binds the threads to places.

FALSE
> Allows threads to be moved between places and disables thread affinity.

MASTER
> Instructs the execution environment to assign the threads in the team to the same place as the master thread.

CLOSE
> Instructs the execution environment to assign the threads in the team to the places that are close to the place of the parent thread. The place partition is not changed by this policy. Each implicit task inherits the *place-partition-var* ICV of the parent implicit task. Suppose T threads in the team are assigned to P places in the parent's place partition, the threads are assigned as follows:
> - If T is less than or equal to P, the master thread executes on the place of the parent thread. The thread with the next smallest thread number executes on the next place in the place partition, and so on, with wrap around with respect to the place partition of the master thread.
> - If T is greater than P, each place contains at least S = floor(T/P) consecutive threads. The first S threads with the smallest thread number (including the master thread) are assigned to the place of the parent thread. The next S threads with the next smallest thread numbers are assigned to the next place in the place partition, and so on, with wrap around with respect to the place partition of the master thread. When P does not divide T evenly, each remaining thread is assigned to a subpartition in the order of the place list.

SPREAD
> Instructs the execution environment to spread a set of T threads as evenly as possible among P places of the parent's place partition at run time. The thread distribution mechanism is as follows:
> - If T is less than or equal to P, the parent partition is divided into T subpartitions, where each subpartition contains at least S=floor(P/T) consecutive places. A single thread is assigned to each subpartition. The master thread executes on the place of the parent thread and is assigned to the subpartition that includes that place. The thread with the next smallest

thread number is assigned to the first place in the next subpartition, and so on, with wrap around with respect to the original place partition of the master thread.

- If T is greater than P, the parent's partition is divided into P subpartitions, where each subpartition contains a single place. Each place contains at least S = floor(T/P) consecutive threads. The first S threads with the smallest thread number (including the master thread) are assigned to the subpartition that contains the place of the parent thread. The next S threads with the next smallest thread numbers are assigned to the next place in the place partition, and so on, with wrap around with respect to the original place partition of the master thread. When P does not divide T evenly, each remaining thread is assigned to a subpartition in the order of the place list.

where

Place

is a hardware unit that holds an unordered set of processors on which one or more threads can execute.

Place list

is an ordered list that describes all places that are available to the applications.

Place partition

is an ordered list that corresponds to a contiguous interval in the place list. The places in the partition are available for a given parallel region.

When **OMP_PROC_BIND** is set to **TRUE**, **MASTER**, **CLOSE**, or **SPREAD**, a place can be assigned with up to THREADS_PER_PLACE threads. Each remaining thread is assigned to a place in the order of the place list.

For each place in **OMP_PLACES**, THREADS_PER_PLACE is a positive integer and is calculated in the following way:

THREADS_PER_PLACE = floor((the number of resources in that place/the total number of resources (including duplicated resources))*OMP_THREAD_LIMIT)

After THREADS_PER_PLACE is calculated for each place in this manner, if the sum of all the THREADS_PER_PLACE values is less than **OMP_THREAD_LIMIT**, each THREADS_PER_PLACE is increased by one, starting from the largest place to the smallest place, until **OMP_THREAD_LIMIT** is reached. Places that are equivalent in size are ordered according to their order in **OMP_PLACES**.

Usage

By default, the **OMP_PROC_BIND** environment variable is not set.

If the initial thread cannot be bound to the first place in the OpenMP place list, the runtime execution environment issues a message and assigns threads according to the default place list.

The **OMP_PROC_BIND** and **XLSMPOPTS** environment variables interact with each other according to the following rules:

Table 7. 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.
	XLSMPOPTS is set to **startproc**/**stride** or **procs**[2].	Threads are bound according to the settings in **XLSMPOPTS**.
	XLSMPOPTS setting is invalid.	Threads are not bound.
OMP_PROC_BIND=TRUE	**XLSMPOPTS** is not set.	Threads are bound.
	XLSMPOPTS is set to **startproc**/**stride** or **procs**[2].	Threads are bound according to the settings in **XLSMPOPTS**[1].
	XLSMPOPTS setting is invalid.	Threads are bound.
OMP_PROC_BIND=FALSE	**XLSMPOPTS** is not set.	Threads are not bound.
	XLSMPOPTS is set to **startproc**/**stride** or **procs**[2].	
	XLSMPOPTS setting is invalid.	

Note:

1. If **procs** is set and the number of CPU IDs specified is smaller than the number of threads that are used by the program, the remaining threads are also bound to the listed CPU IDs but not in any particular order. If **XLSMPOPTS=startproc** is used, the value specified by **startproc** is smaller than the number of CPUs, and the value that is specified by **stride** causes a thread to bind to a CPU outside the range of available places, some of the threads are bound and some are not.

2. The **startproc**/**stride** and **procs** suboptions of **XLSMPOPTS** are deprecated.

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 places. If portability of your application is important, use only the **OMP_PROC_BIND** environment variable to control thread binding.

When **OMP_PROC_BIND** is set to **MASTER**, **CLOSE**, or **SPREAD**, the suboption settings **startproc**/**stride** or **procs** of **XLSMPOPTS** are ignored.

For a program that contains both OpenMP and OpenMPI code, the OpenMP runtime detects the existence of OpenMPI code by the presence of the **OMPI_COMM_WORLD_RANK** environment variable. If you do not set **OMP_PLACES** explicitly, the compiler sets **OMP_PROC_BIND** to be **TRUE**.

Examples

The following examples demonstrate the thread binding and thread affinity results when you compile myprogram.c with different environment variable settings.

myprogram.c

```
int main(){
    // ...
}
```

Environment variable settings 1

```
OMP_NUM_THREADS=4;
OMP_PROC_BIND=MASTER;
OMP_PLACES='{0:4},{4:4},{8:4},{12:4},{16:4},{20:4},{24:4},{28:4}'
```

Results 1: Every thread in the team is assigned to the place on which the master executes. Four threads are assigned to place 0.

Environment variable settings 2

```
OMP_NUM_THREADS=4;
OMP_PROC_BIND=close;
OMP_PLACES='{0:4},{4:4},{8:4},{12:4},{16:4},{20:4},{24:4},{28:4}'
```

Results 2: The thread is assigned to a place that is close to the place of the parent thread. The thread assignment is as follows:

- OMP thread 0 is assigned to place 0
- OMP thread 1 is assigned to place 1
- OMP thread 2 is assigned to place 2
- OMP thread 3 is assigned to place 3

Environment variable settings 3

```
OMP_NUM_THREADS=4;
OMP_PROC_BIND=spread;
OMP_PLACES='{0:4},{4:4},{8:4},{12:4},{16:4},{20:4},{24:4},{28:4}'
```

Results 3: The number of threads 4 is smaller than the number of places 8, so four subpartitions are formed. 8 is evenly divided by 4, so the thread assignment is as follows:

- OMP thread 0 is assigned to place 0
- OMP thread 1 is assigned to place 2
- OMP thread 2 is assigned to place 4
- OMP thread 3 is assigned to place 6

Environment variable settings 4

```
OMP_NUM_THREADS=5;
OMP_PROC_BIND=spread;
OMP_PLACES='{0:4},{4:4},{8:4},{12:4},{16:4},{20:4},{24:4},{28:4}'
```

Results 4: The number of threads 5 is smaller than the number of places 8, so five subpartitions are formed. 8 is not evenly divided by 5, so threads are assigned to the places in order. The thread assignment is as follows:

- OMP thread 0 is assigned to place 0
- OMP thread 1 is assigned to place 2
- OMP thread 2 is assigned to place 4
- OMP thread 3 is assigned to place 6
- OMP thread 4 is assigned to place 7

Environment variable settings 5

```
OMP_NUM_THREADS=8;
OMP_PROC_BIND=spread;
OMP_PLACES='{0:4},{4:4},{8:4},{12:4}'
```

Results 5: The number of threads 8 is greater than the number of places 4, so four subpartitions are formed. 8 is evenly divided by 4, so two threads are assigned to each subpartition. The thread assignment is as follows:

- OMP thread 0 and thread 1 are assigned to place 0
- OMP thread 2 and thread 3 are assigned to place 1
- OMP thread 4 and thread 5 are assigned to place 2

- OMP thread 6 and thread 7 are assigned to place 3

Environment variable settings 6

```
OMP_NUM_THREADS=7;
OMP_PROC_BIND=spread;
OMP_PLACES='{0:4},{4:4},{8:4},{12:4}'
```

Results 6: The number of threads 7 is greater than the number of places 4, so four subpartitions are formed. 7 is not evenly divided by 4, so one thread (`floor(7/4)=1`) is assigned to each subpartition. The thread assignment is as follows:

- OMP thread 0 is assigned to place 0
- OMP thread 1 and thread 2 are assigned to place 1
- OMP thread 3 and thread 4 are assigned to place 2
- OMP thread 5 and thread 6 are assigned to place 3

Related reference:

"omp_get_proc_bind" on page 556

Related information:

"XLSMPOPTS" on page 20

"OMP_PLACES" on page 30

OMP_SCHEDULE: The **OMP_SCHEDULE** environment variable specifies the schedule type used for loops that are explicitly assigned to runtime schedule type with the **OpenMP schedule** clause.

For example:

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

Valid options for schedule type are:

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

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

The default schedule type is **auto**.

Related reference:

"omp_set_schedule" on page 563

"omp_get_schedule" on page 557

OMP_STACKSIZE:
The **OMP_STACKSIZE** environment variable specifies the size of the stack for threads created by the OpenMP run time. The syntax is as follows:

```
►►──OMP_STACKSIZE=──┬─size──┬──────────────────────────────►◄
                    ├─sizeB─┤
                    ├─sizeK─┤
                    ├─sizeM─┤
                    └─sizeG─┘
```

size
>   is a positive integer that specifies the size of the stack for threads that are
>   created by the OpenMP run time.

"B", "K", "M", "G"
>   are letters that specify whether the given size is in Bytes, Kilobytes, Megabytes,
>   or Gigabytes.

If only size is specified and none of **"B", "K", "M", "G"** is specified, size is in
Kilobytes by default. This environment variable does not control the size of the
stack for the initial thread.

The value assigned to the **OMP_STACKSIZE** environment variable is case
insensitive and might have leading and trailing white space. The following
examples show how you can set the **OMP_STACKSIZE** environment variable.

```
export OMP_STACKSIZE="10M"
export OMP_STACKSIZE=" 10 M "
```

If the value of **OMP_STACKSIZE** is not set, the initial value is set to the default
value. (up to the limit that is imposed by system resources).

If the compiler cannot deliver the stack size specified by the environment variable,
or if **OMP_STACKSIZE** does not conform to the valid format, the compiler sets
the environment variable to the default value.

The **OMP_STACKSIZE** environment variable takes precedence over the **stack**
suboption of the **XLSMPOPTS** environment variable.

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

►►──OMP_THREAD_LIMIT──=──*n*──►◄

n   The number of OpenMP threads to use for the whole program. It must be a
>   positive scalar integer that is less than 65536.

Usage

When **OMP_THREAD_LIMIT**=1, the parallel regions are run sequentially rather
than in parallel. However, when **OMP_THREAD_LIMIT** is much smaller than the
number of threads that are required in the program, the parallel region might still
run in parallel but with fewer threads. When there are nested parallel regions,
some parallel regions might run in parallel, some might run sequentially, and some
might run in parallel but with threads that are recycled from other regions.

If **OMP_THREAD_LIMIT** is not defined and **OMP_NESTED=TRUE**, the default
value of **OMP_THREAD_LIMIT** is the greater value of either the multiplication of
all **OMP_NUM_THREADS** levels or the number of total resources in
OMP_PLACES.

If **OMP_THREAD_LIMIT** is not defined and **OMP_NESTED=FALSE**, the default
value of **OMP_THREAD_LIMIT** is the greater value of either the first level of
OMP_NUM_THREADS or the number of total resources in **OMP_PLACES**.

If neither **OMP_THREAD_LIMIT** nor **OMP_NESTED** is defined, the default value of **OMP_THREAD_LIMIT** is the number of total resources in **OMP_PLACES**.

Examples

Suppose **OMP_THREAD_LIMIT** is not defined and **OMP_PLACES**={0,1,2,3,4,5,6,7},{8,9,10,11,12,13,14,15}. The number of total resources in **OMP_PLACES** is 16.

Example 1

When **OMP_NESTED=TRUE** and **OMP_NUM_THREADS**=2,12, the default value of **OMP_THREAD_LIMIT** is 24, because the multiplication of all **OMP_NUM_THREADS** levels is 24 and 24 is greater than 16.

Example 2

When **OMP_NESTED=FALSE** and **OMP_NUM_THREADS**=2,4, the default value of **OMP_THREAD_LIMIT** is 16, because the first level of **OMP_NUM_THREADS** is 2 and 16 is greater than 2.

Related information:

OMP_WAIT_POLICY:
The **OMP_WAIT_POLICY** environment variable provides hints about the preferred behavior of waiting threads during program execution. The syntax is as follows:

```
                          ┌─PASSIVE─┐
►►──OMP_WAIT_POLICY=──┴─ACTIVE──┴──────────────────────────────────►◄
```

Use **ACTIVE** if you want waiting threads to mostly be active. That is, the threads consume processor cycles while waiting. For example, waiting threads can spin while waiting. The **ACTIVE** wait policy is recommended for maximum performance on the dedicated machine.

Use **PASSIVE** if you want waiting threads to mostly be passive. That is, the threads do not consume processor cycles while waiting. For example, waiting threads can sleep or yield the processor to other threads.

The default value of **OMP_WAIT_POLICY** is **PASSIVE**.

Note: If you set the **OMP_WAIT_POLICY** environment variable and specify the `spins`, `yields`, or `delays` suboptions of the **XLSMPOPTS** environment variable, **OMP_WAIT_POLICY** takes precedence.

Using custom compiler configuration files

The XL C/C++ compiler generates a default configuration file /opt/ibm/xlC/13.1.6/etc/xlc.cfg.*$OSRelease*.gcc*$gccVersion* 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. Examples of the default configuration file are listed below:

- /opt/ibm/xlC/13.1.6/etc/xlc.cfg.sles.12.gcc.4.8.2
- /opt/ibm/xlC/13.1.6/etc/xlc.cfg.rhel.7.3.gcc.4.8.5
- /opt/ibm/xlC/13.1.6/etc/xlc.cfg.centos.7.gcc.4.8.3
- /opt/ibm/xlC/13.1.6/etc/xlc.cfg.ubuntu.16.04.gcc.4.8.2

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 **xlc** compiler invocation command. This is to avoid forcing your users to specify this option on the command line for every compilation, because **-qnolist** is automatically in effect every time the compiler is called with the **xlc** 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 reference:

"-F" on page 77

Related information:

"Compile-time and link-time environment variables" on page 18

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

Figure 1. Sample configuration file

In this example:
- stanza A uses option sets *A* and *Z*
- stanza B uses option sets *B1*, *B2*, *D*, *A*, and *Z*
- stanza C uses option sets *C*, *A*, and *Z*
- stanza D uses option sets *D*, *A*, and *Z*

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

By default, values defined in a stanza in a configuration file are added to the list of values specified in previously processed stanzas. For example, assume that the XLC_USR_CONFIG environment variable is set to point to the user-defined configuration file at ~/userconfig1. With the user-defined and default configuration files shown in the following example, the compiler references the **xlc** 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*.

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

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

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

DEFLT:
    options=<C>
```

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

Figure 3. Default configuration file xlc.cfg

Overriding the default order of attribute values

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

Table 8. Assignment operators and attribute ordering

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

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

Custom user-defined configuration file ~/userconfig2

Default configuration file xlc.cfg

```
xlc_prepend: use=xlc
          options-=<B1>
xlc_replace: use=xlc
          options:=<B2>
xlc_append: use=xlc
          options+=<B3>

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

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

DEFLT:
    options=<C>
```

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

1. stanza xlc uses *B, D*, and *C*
2. stanza xlc_prepend uses *B1, B, D*, and *C*
3. stanza xlc_replace uses *B2*
4. stanza xlc_append uses *B, D, C*, and *B3*

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

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

Figure 4. Using additional assignment operations

Examples of stanzas in custom configuration files

`DEFLT: use=DEFLT` `    options = -g`	This example specifies that the **-g** option is to be used in all compilations.
`xlc: use=xlc    options+=-qlist`	This example specifies that **-qlist** is to be used for any compilation called by the **xlc** command. This **-qlist** specification overrides the default setting of **-qlist** specified in the system configuration file.
`DEFLT: use=DEFLT` `    libraries=-L/home/user/lib,-lmylib`	This example specifies that all compilations should link with /home/user/lib/libmylib.a.

Using IBM XL C/C++ for Linux, V13.1.6 with the Advance Toolchain

IBM XL C/C++ for Linux, V13.1.6 supports IBM Advance Toolchain 11.0, which is a set of open source development tools and runtime libraries. With IBM Advance Toolchain 11.0, you can take advantage of the latest POWER® hardware features on Linux, especially the tuned libraries. For more information about the Advance Toolchain 11.0, see the Advance toolchain for Linux on Power website.

To use IBM XL C/C++ for Linux, V13.1.6 with the Advance Toolchain, take the following steps:

1. Install the **at11.0** packages into the default installation location. For instructions, see Installation on the Advance toolchain for Linux on Power website.

2. Run the **xlc_configure** utility to create the **xlc.at.cfg** configuration file. In the **xlc.at.cfg** configuration file, all other entities except the XL C/C++ compiler are directed to those of the Advance Toolchain. The entities include the linker, headers, and runtime libraries.

   Note: To run the **xlc_configure** utility, you must either become the root user or use the sudo command.

   `xlc_configure -at`

3. Invoke the XL compiler with the Advance Toolchain support.
   - If you installed the compiler in the default location, issue the following commands:

     `/opt/ibm/xlC/13.1.6/bin/xlc_at`
     `/opt/ibm/xlC/13.1.6/bin/xlC_at`

   - If you installed the compiler in an NDI location, issue the following commands:

     `$ndi_path/xlC/13.1.6/bin/xlc_at`
     `$ndi_path/xlC/13.1.6/bin/xlC_at`

Note: If you use the XL compiler with the Advance Toolchain support to build your application, your application can run only under the Advance Toolchain environment because the application depends on the runtime library of the Advance Toolchain. If you copy the application to run on other machines, ensure that the Advance Toolchain, or at least the runtime library of the Advance Toolchain, is available on those machines.

Editing the default configuration file

The configuration file specifies information that the compiler uses when you invoke it. XL C/C++ provides the default configuration file /opt/ibm/xlC/13.1.6/etc/xlf.cfg at installation time.

If you want many users to be able to choose among several sets of compiler options, you may want to add new named stanzas to the configuration file and to create new commands that are links to existing commands. For example, you could specify something similar to the following to create a link to the **c99** command:

```
ln -s /opt/ibm/xlC/13.1.6/bin/c99 /home/username/bin/c99
```

When you run the compiler under another name, it uses whatever options, libraries, and so on, that are listed in the corresponding stanza.

Notes:

* The configuration file contains other named stanzas to which you may want to link.
* If you make any changes to the configuration file and then move or copy your makefiles to another system, you will also need to copy the changed configuration file.
* You cannot use tabs as separator characters in the configuration file. If you modify the configuration file, make sure that you use spaces for any indentation.

Configuration file attributes

The configuration file contains the following attributes:

__GNUC__
> The version of **gcc**.

__GNUC_MINOR__
> The release of **gcc**.

__GNUC_PATCHLEVEL__
> The modification level of **gcc**.

as_64 The absolute path name of the assembler.

bolt The absolute path name of the binder.

ccomp The absolute path name of the compiler front end.

code The absolute path name of the optimizing code generator.

cppfilt
> The absolute path name of the C++ demangler.

crt_64 The path name of the object file which contains the startup code. This object file is passed as the first parameter to the linkage editor.

defaultmsg
> The absolute path name of the default message files.

dis The absolute path name of the disassembler.

gcc_path_64
Specifies the path to the 64-bit tool chain.

gcrt_64
Same as **crt_64**, but the object file contains profiling code for the **-pg** option.

genhtml
Specifies the path to the **genhtml** utility.

ipa The absolute path name of the program that performs interprocedural optimizations, loop optimizations, and program parallelization.

ld_64 The absolute path name of the linker.

ldopt Lists names of options that are assumed to be linker options for cases where, for example, a compiler option and a linker option use the same letter. The list is a concatenated set of single-letter flags. Any flag that takes an argument is followed by a colon, and the whole list is enclosed by double quotation marks.

You might find it more convenient to set up this attribute than to pass options to the linker through the **-W** compiler option. However, most unrecognized options are passed to the linker anyway.

list The absolute path name of the lister.

listlibs
The path to libraries for listing support.

mcrt_64
Same as for **crt_64**, but the object file contains profiling code for the **-p** option.

options
A string of options that are separated by commas. The compiler processes these options as if you entered them on the command line before any other option. This attribute lets you shorten the command line by including commonly used options in one central place.

os_major
The version of the operating system.

os_minor
The release of the operating system.

os_patchlevel
The modification level of the operating system.

os_variant
The value of **$ID** from the /etc/os-release file.

slm_auth
The path of the authorization file. The default is **/etc/XLAuthorizedUsers**.

slm_dir
The directory of the SLM tag file. The default is **/var/opt/ibm/xl-compiler/** for a default installation, or *$prefix*/**var/opt/ibm/xl-compiler/** for a nondefault installation, where *$prefix* is the nondefault installation path.

slm_period
The number of seconds that each metric covers. The SLM daemon outputs the usage information for every defined period of time. The default is **300**.

slm_limit

> The maximum number of bytes that each tag file is allowed to occupy. The default is **5000000**.

slm_timeout

> The minimum number of seconds that the daemon must wait before terminating. The default is **5**.

smplibraries

> Specifies the libraries that are used to link programs that you compiled with the **-qsmp** compiler option.

transforms

> The absolute path name of the transformation report listing.

xlCcopt

> The options for compiling C code with the compiler.

xlc_c_complexgccinc

> The GNU compiler **complexgcc** include path.

xlc_c_stdinc

> The compiler include path for C code.

xlc_cpp_stdinc

> The compiler include path for C++ code.

xlc_path

> The location of the product.

xslt   The absolute path name of the XSLT processor.

Notes:

- To specify multiple search paths for compilation include files, separate each path location with a comma as follows:

  ```
  include = -I/path1, -I/path2, ...
  ```

- You can use the "-F" on page 77 option to select a different configuration file, a specific stanza in the configuration file, or both.

Related information

- "Types of input files" on page 3
- "Types of output files" on page 4
- Chapter 3, "Tracking compiler license usage," on page 47

Chapter 3. Tracking compiler license usage

You can enable IBM Software License Metric (SLM) Tags logging to track compiler license usage. This information can help you determine whether your organization's use of the compiler exceeds your compiler license entitlements.

Understanding compiler license tracking

You can enable IBM Software License Metric (SLM) Tags logging in the compiler so that IBM License Metric Tool (ILMT) can track compiler license usage.

Types of compiler licenses logged

The compiler logs the usage of the following two types of compiler licenses:
- **Authorized user licenses**: Each compiler license is tied to a specific user ID, designated by that user's uid.
- **Concurrent user licenses**: A certain number of concurrent users are authorized to use the compiler license at any given time.

The compiler logs each compiler invocation as either a concurrent user or an authorized user invocation, depending on the presence of the invoking user's uid in a file that lists the authorized users.

The SLM daemon process

The SLM tag file is created by the SLM daemon. The SLM daemon is a regular executable in the compiler. It outputs a tag file for every user-defined period of time. The daemon terminates if no user invokes the compiler within a user-defined period of time, and restarts with a new invocation of the compiler. You can define both time periods in the configuration file.

The SLM tag file

The compiler logs compiler license usage to an SLM tag file *slm_dir*/*hash*.slmtags, where *slm_dir* is a configurable directory, and *hash* is the MD5 hash of the SWID and root directory of the current product. The default value of *slm_dir* is /var/opt/ibm/xl-compiler/ for a default installation, or *$prefix*/var/opt/ibm/xl-compiler for a nondefault installation, where *$prefix* is the nondefault installation path. The *slm_dir* directory must be readable and writable by all users invoking the compiler.

The main elements of the tag file are as follows:

\<StartTime\> and \<EndTime\>
> The start time and end time of the invocations.

\<Value\>
> The maximum number of users that invoked the compiler simultaneously within the time period denoted by the **\<StartTime\>** and **\<EndTime\>** elements.

\<Type\>
> The type of compiler license logged.

47

\<Metric\>

> All the preceding elements are contained in the **\<Metric\>** element. The SLM daemon outputs up to two **\<Metric\>** elements for every time period, one for authorized users and one for concurrent users.

The following example is a tag file generated by the compiler. It logs three concurrent user invocations followed by one authorized user invocation.

```
<SchemaVersion>2.1.1</SchemaVersion>
 <SoftwareIdentity>
        <PersistentId>43d3e5201c664350a0cb3a4772381fe0</PersistentId>
        <Name>IBM XL
C/C++ for Linux</Name>
        <InstanceId>/opt/ibm/xlC/13.1.6</InstanceId>
 </SoftwareIdentity>
<Metric logTime="2016-04-01T18:39:51Z">
        <Type>AUTHORIZED_USER</Type>
        <Value>1</Value>
        <Period>
                <StartTime>2016-04-01T18:34:51Z</StartTime>
                <EndTime>2016-04-01T18:39:51Z</EndTime>
        </Period>
</Metric>
<Metric logTime="2016-04-01T18:39:51Z">
        <Type>CONCURRENT_USER</Type>
        <Value>3</Value>
        <Period>
                <StartTime>2016-04-01T18:14:51Z</StartTime>
                <EndTime>2016-04-01T18:19:51Z</EndTime>
        </Period>
</Metric>
```

The tag file has a configurable size limit and the default is 5,000,000 bytes. If the size of the current file exceeds the maximum size, old **\<Metric\>** elements in the file are removed.

Error logs

If an error occurs before the daemon forks, a return code of -1 is returned. Otherwise, no failure will be reported. In either case, the error will be logged to *slm_dir/.hash/log*.

Related reference:

-qslmtags

Related information:

Configuring file attributes

Setting up SLM Tags logging

To enable ILMT to track compiler usage, you must set up SLM Tags logging.

About this task

If your compiler license is an authorized user license, use these steps to set up XL compiler SLM Tags logging.

Procedure

1. Determine the user IDs of your authorized users.
2. Create a file with the name XLAuthorizedUsers in the /etc directory. You can change the location of the XLAuthorizedUsers file by specifying the **slm_auth**

attribute of the configuration file. The file contains the information for authorized users, one line for each user. Each line contains the numeric uid of the authorized user followed by a comma, and the Software ID (SWID) of the authorized product.

You can obtain the uid of a user ID by using the **id -u username** command, where you replace *username* with the user ID you are looking up.

You can find the SWID of the product by running the following command:

```
grep persistentId /opt/ibm/xlC/V.R.M/swidtag/*.swidtag
```

where *V.R.M* is the Version.Release.Modification level of the compiler that is installed on the system.

For IBM XL C/C++ for Linux, the SWID is `43d3e5201c664350a0cb3a4772381fe0`, which does not change across compiler versions or for different installation instances.

3. Set `/etc/XLAuthorizedUsers` to be readable by all users invoking the compiler:

```
chmod a+r /etc/XLAuthorizedUsers
```

Results

If a user's uid is listed in `/etc/XLAuthorizedUsers`, the compiler will log an authorized user invocation along with the SWID of the compiler being used. Otherwise the compiler will log a concurrent user invocation.

Note that XL compiler SLM Tags logging does not enforce license compliance. It only logs compiler invocations so that you can use the collected data and IBM License Metric Tool to determine whether your use of the compiler is within the terms of your compiler license.

Example

Suppose that you have three authorized users whose IDs are bsmith, rsingh, and jchen. For these user IDs you enter the following commands and see the corresponding output in a command shell:

```
$id -u bsmith
24461
$id -u rsingh
9204
$id -u jchen
7531
```

You run the following command to obtain the SWID:

```
$ grep persistentId /opt/ibm/xlC/13.1.6/swidtag/*.swidtag
<Meta persistentId="43d3e5201c664350a0cb3a4772381fe0"/>
```

Then you create `/etc/XLAuthorizedUsers` with the following lines to authorize these users to use the compiler:

```
24461,43d3e5201c664350a0cb3a4772381fe0
9204,43d3e5201c664350a0cb3a4772381fe0
7531,43d3e5201c664350a0cb3a4772381fe0
```

Related reference:

-qslmtags

Related information:

IBM License Metric Tool (ILMT)

Configuration file attributes

Chapter 4. Compiler options reference

This section contains a summary of the compiler options available in XL C/C++ by functional category, followed by detailed descriptions of the individual options. It also provides a list of supported GCC options.

Related information
- "Specifying compiler options" on page 5

Summary of compiler options by functional category

The XL C/C++ options available on the Linux 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 52
- "Language element control" on page 53
- "Template control (C++ only)" on page 54
- "Floating-point and integer control" on page 54
- "Error checking and debugging" on page 56
- "Listings, messages, and compiler information" on page 60
- "Optimization and tuning" on page 60
- "Object code control" on page 55
- "Linking" on page 63
- "Portability and migration" on page 64
- "Compiler customization" on page 64

Output control

The options in this category control the type of output file the compiler produces, as well as the locations of the output. These are the basic options that determine the following aspects:

- The compiler components that will be invoked

- The preprocessing, compilation, and linking steps that will (or will not) be taken

- The kind of output to be generated

Table 9. Compiler output options

Option name	Description
"-c" on page 91	Instructs the compiler to compile or assemble the source files only but do not link. With this option, the output is a .o file for each source file.
"-C, -C!" on page 73	When used in conjunction with the **-E** or **-P** options, preserves or removes comments in preprocessed output.
"-dM (-qshowmacros)" on page 92	Emits macro definitions to preprocessed output.
"-E" on page 75	Preprocesses the source files named in the compiler invocation, without compiling.

Option name	Description
"-o" on page 134	Specifies a name for the output object, assembler, executable, or preprocessed file.
"-P" on page 84	Preprocesses the source files named in the compiler invocation, without compiling, and creates an output preprocessed file for each input file.
"-qmakedep, -MD (-qmakedep=gcc)" on page 175	Produces the dependency files that are used by the **make** tool for each source file.
"-qtimestamps" on page 216	Controls whether or not implicit time stamps are inserted into an object file.
"-shared (-qmkshrobj)" on page 220	Creates a shared object from generated object files.
"-S" on page 86	Generates an assembler language file for each source file.
"-X (-W)" on page 88	**-Xpreprocessor** *option* or **-Wp,***option* passes the listed option directly to the preprocessor.

The following options are supported by XL C/C++ for GCC compatibility. For details about these options, see the GNU Compiler Collection online documentation at http://gcc.gnu.org/onlinedocs/.

- -###
- -d*CHARS*
- -M
- -MD
- -MF *file*
- -MG
- -MM
- -MMD
- -MP
- -MQ *target*
- -MT *target*
- -Xpreprocessor

Input control

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

Table 10. Compiler input options

Option name	Description
"-include (-qinclude)" on page 122	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.
"-I" on page 78	Adds a directory to the search path for include files.

Table 10. Compiler input options (continued)

Option name	Description
"-qidirfirst" on page 155	Searches for user included files in directories that are specified by the **-I** option before searching any other directories.
"-qstdinc, -qnostdinc (-nostdinc, -nostdinc++)" on page 206	Specifies whether the standard include directories are included in the search paths for system and user header files.
"-x (-qsourcetype)" on page 231	Instructs the compiler to treat all recognized source files as a specified source type, regardless of the actual file name suffix.

Language element control

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

Table 11. Language element control options

Option name	Description
"-D" on page 74	Defines a macro as in a `#define` preprocessor directive.
"-fasm (-qasm)" on page 94	Controls the interpretation and subsequent generation of code for assembler language extensions.
"-maltivec (-qaltivec)" on page 129	Enables the compiler support for vector data types and operators.
"-fdollars-in-identifiers (-qdollar)" on page 97	Allows the dollar-sign ($) symbol to be used in the names of identifiers.
"-qstaticinline (C++ only)" on page 205	Controls whether inline functions are treated as having `static` or `extern` linkage.
"-std (-qlanglvl)" on page 224	Determines whether source code and compiler options should be checked for conformance to a specific language standard, or subset or superset of a standard.
"-U" on page 87	Undefines a macro defined by the compiler or by the **-D** compiler option.
"-X (-W)" on page 88	**-Xassembler** *option* or **-Wa,***option* passes the listed option directly to the assembler.

The following options are supported by XL C/C++ for GCC compatibility. For details about these options, see the GNU Compiler Collection online documentation at http://gcc.gnu.org/onlinedocs/.

- -ansi
- -fconstexpr-depth
- -fconstexpr-steps
- -ffreestanding
- -fgnu89-inline

- -fhosted
- -fno-access-control
- -fno-builtin
- -fno-gnu-keywords
- -fno-operator-names
- -fno-rtti
- -fpermissive
- -fsigned-char
- -ftemplate-backtrace-limit
- -ftemplate-depth
- -funsigned-char
- -trigraphs
- -Xassembler

Template control (C++ only)

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

Table 12. C++ template options

Option name	Description
"-ftemplate-depth (-qtemplatedepth) (C++ only)" on page 110	Specifies the maximum number of recursively instantiated template specializations that will be processed by the compiler.
"-qtmplinst (C++ only)" on page 217	Manages the implicit instantiation of templates.

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. Use the options in the following table to control trade-offs between floating-point performance and adherence to IEEE standards.

Table 13. Floating-point and integer control options

Option name	Description
"-fsigned-char, -funsigned-char (-qchars)" on page 106	Determines whether all variables of type char is treated as signed or unsigned.
"-qfloat" on page 147	Selects different strategies for speeding up or improving the accuracy of floating-point calculations.
"-qstrict" on page 208	Ensures that optimizations that are done by default at the **-03** and higher optimization levels, and, optionally at **-02**, do not alter the semantics of a program.
"-y" on page 232	Specifies the rounding mode for the compiler to use when evaluating constant floating-point expressions at compile time.

Object code control

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

Table 14. Object code control options

Option name	Description
"-fcommon (-qcommon)" on page 95	Controls where uninitialized global variables are allocated.
"-fexceptions (-qeh) (C++ only)" on page 98	The **-fexceptions** option controls whether exception handling is allowed in the module being compiled. The **-qeh** option controls whether exception handling is enabled in the module being compiled.
"-qfuncsect" on page 152	Places instructions for each function in a separate section. Placing each function in its own section might reduce the size of your program because the linker can collect garbage per function rather than per object file.
"-qinlglue" on page 159	When used with **-02** or higher optimization, inlines glue code that optimizes external function calls in your application.
"-qpriority (C++ only)" on page 187	Specifies the priority level for the initialization of static objects.
"-qreserved_reg" on page 190	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 192	Specifies the storage type for string literals.
"-qroconst" on page 193	Specifies the storage location for constant values.
"-qrtti, -fno-rtti (-qnortti) (C++ only)" on page 194	Generates runtime type identification (RTTI) information for exception handling and for use by the `typeid` and `dynamic_cast` operators.
"-qsaveopt" on page 195	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.
▶ **GPU** -qtgtarch **GPU** ◀	Specifies real or virtual GPU architectures where the code may run. This allows the compiler to take maximum advantage of the capabilities and machine instructions that are specific to a GPU architecture, or common to a virtual architecture.

Table 14. Object code control options (continued)

Option name	Description
"-r" on page 219	Produces a nonexecutable output file to use as an input file in another ld command call. This file may also contain unresolved symbols.
"-s" on page 220	Strips the symbol table, line number information, and relocation information from the output file.

The following options are supported by XL C/C++ for GCC compatibility. For details about these options, see the GNU Compiler Collection online documentation at http://gcc.gnu.org/onlinedocs/.

- -fpack-struct
- -fPIC, -fno-PIC
- -fPIE, -fno-PIE
- -fsemantic-interposition, -fno-semantic-interposition
- -fshort-wchar

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 60.

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

Table 15. Error checking and debugging options

Option name	Description
"-### (-#) (pound sign)" on page 66	Previews the compilation steps specified on the command line, without actually invoking any compiler components.
-fstack-protector (-qstackprotect)	Provides protection against malicious input data or programming errors that overwrite or corrupt the stack.
"-fstandalone-debug" on page 107	When used with the **-g** option, controls whether to generate the debugging information for all symbols.
"-fsyntax-only (-qsyntaxonly)" on page 109	Performs syntax checking without generating an object file.
"-g" on page 120	Generates debugging information for use by a symbolic debugger, and makes the program state available to the debugging session at selected source locations.

Option name	Description
"-qcheck" on page 141	Generates code that performs certain types of runtime checking.
"-ftrapping-math (-qflttrap)" on page 111	Determines what types of floating-point exceptions to detect at run time.
"-qfulldebug" on page 150	Generates the debugging information for all class members.
"-qfullpath" on page 151	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.
"-qinitauto" on page 157	Initializes uninitialized automatic variables to a specific value, for debugging purposes.
"-qkeepparm" on page 167	When used with **-02** or higher optimization, specifies whether procedure parameters are stored on the stack.
"-qlinedebug" on page 169	Generates only line number and source file name information for a debugger.
"-Werror (-qhalt)" on page 90	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.
"-Wunsupported-xl-macro" on page 91	Checks whether any unsupported XL macro is used.

Options to control diagnostic messages formatting

The following options are supported by XL C/C++ for GCC compatibility. For details about these options, see the GNU Compiler Collection online documentation at http://gcc.gnu.org/onlinedocs/.

- -fansi-escape-codes
- -fcolor-diagnostics
- -fdiagnostics-format=[clang|msvc|vi]
- -fdiagnostics-fixit-info
- -fdiagnostics-print-source-range-info
- -fdiagnostic-parsable-fixits
- -fdiagnostic-show-category=[none|id|name]
- -fdiagnostics-show-name
- -fdiagnostic-show-template-tree
- -fmessage-length
- -fno-diagnostics-show-caret
- -fno-diagnostics-show-option
- -fno-elide-type

- -fshow-column
- -fshow-source-location
- -pedantic
- -pedantic-errors
- -Wambiguous-member-template
- -Wbind-to-temporary-copy
- -Wextra-tokens

Options to request or suppress warnings

The following options are supported by XL C/C++ for GCC compatibility. For details about these options, see the GNU Compiler Collection online documentation at http://gcc.gnu.org/onlinedocs/.

- -fsyntax-only
- -w
- -Wall
- -Wbad-function-cast
- -Wcast-align
- -Wchar-subscripts
- -Wcomment
- -Wconversion
- -Wc++11-compat
- -Wdelete-non-virtual-dtor
- -Wempty-body
- -Wenum-compare
- -Werror=foo
- -Weverything
- -Wfatal-errors
- -Wfloat-equal
- -Wfoo
- -Wformat
- -Wformat=n
- -Wformat=2
- -Wformat-nonliteral
- -Wformat-security
- -Wformat-y2k
- -Wignored-qualifiers
- -Wimplicit-int
- -Wimplicit-function-declaration
- -Wimplicit
- -Wmain
- -Wmissing-braces
- -Wmissing-field-initializers
- -Wmissing-prototypes
- -Wnarrowing
- -Wno-attributes

- -Wno-builtin-macro-redefined
- -Wno-deprecated
- -Wno-deprecated-declarations
- -Wno-division-by-zero
- -Wno-endif-labels
- -Wno-format
- -Wno-format-extra-args
- -Wno-format-zero-length
- -Wno-int-conversion
- -Wno-invalid-offsetof
- -Wno-int-to-pointer-cast
- -Wno-multichar
- -Wnonnull
- -Wno-return-local-addr
- -Wno-unused-result
- -Wno-virtual-move-assign
- -Wnon-virtual-dtor
- -Woverlength-strings
- -Woverloaded-virtual
- -Wpedantic -pedantic -pedantic-errors
- -Wpadded
- -Wparantheses
- -Wpointer-arith
- -Wpointer-sign
- -Wreorder
- -Wreturn-type
- -Wsequence-point
- -Wshadow
- -Wsign-compare
- -Wsign-conversion
- -Wsizeof-pointer-memaccess
- -Wstack-protector
- -Wswitch
- -Wsystem-headers
- -Wtautological-compare
- -Wtype-limits
- -Wtrigraphs
- -Wundef
- -Wuninitialized
- -Wunknown-pragmas
- -Wunused
- -Wunused-label
- -Wunused-parameter
- -Wunused-variable
- -Wunused-value

- -Wvariadic-macros
- -Wvarargs
- -Wvla
- -Wwrite-strings

Listings, messages, and compiler information

The options in this category allow your 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 56 to provide a more robust overview of your application when checking for errors and unexpected behavior.

Table 16. Listings and messages options

Option name	Description
"-fdump-class-hierarchy (-qdump_class_hierarchy) (C++ only)" on page 97	Dumps a representation of the hierarchy and virtual function table layout of each class object to a file.
"-qlist" on page 170	Produces a compiler listing file that includes object and constant area sections.
"-qlistfmt" on page 171	Creates a report in XML or HTML format to help you find optimization opportunities.
"-qreport" on page 188	Produces listing files that show how sections of code have been optimized.
"-qslmtags" on page 199	Controls whether SLM Tags logging tracks compiler license usage.
"--help (-qhelp)" on page 68	Displays the man page of the compiler.
"--version (-qversion)" on page 68	Displays the version and release of the compiler being invoked.

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 among an increase in compile time, a reduction in debugging capability, and the improvements that optimization can provide.

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

Table 17. Optimization and tuning options

Option name	Description
"-finline-functions (-qinline)" on page 99	Attempts to inline functions instead of generating calls to those functions, for improved performance.

Table 17. Optimization and tuning options (continued)

Option name	Description
"-fstrict-aliasing (-qalias=ansi), -qalias" on page 108	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.
"-funroll-loops (-qunroll), -funroll-all-loops (-qunroll=yes)" on page 116	Controls loop unrolling, for improved performance. **Equivalent pragma:** #pragma unroll
"-fvisibility (-qvisibility)" on page 118	Specifies the visibility attribute for external linkage entities in object files. The external linkage entities have the visibility attribute that is specified by the **-fvisibility** option if they do not get visibility attributes from pragma directives, explicitly specified attributes, or propagation rules. **Equivalent pragma:** #pragma GCC visibility push, #pragma GCC visibility pop
"-mcpu (-qarch)" on page 131	Specifies the processor architecture for which the code (instructions) should be generated.
-mtune (-qtune)	Tunes instruction selection, scheduling, and other architecture-dependent performance enhancements to run best on a specific hardware architecture. Allows specification of a target SMT mode to direct optimizations for best performance in that mode.
"-O, -qoptimize" on page 80	Specifies whether to optimize code during compilation and, if so, at which level.
▶ GPU ◀ "-qoffload" on page 177	Enables support for offloading OpenMP target regions to an NVIDIA GPU.
"-p, -pg, -qprofile" on page 135	Prepares the object files produced by the compiler for profiling.
"-qaggrcopy" on page 136	Enables destructive copy operations for structures and unions.
"-qcache" on page 138	Specifies the cache configuration for a specific execution machine.
"-qcompact" on page 143	Avoids optimizations that increase code size.
"-qdataimported, -qdatalocal, -qtocdata" on page 145	Marks data as local or imported.
"-qdirectstorage" on page 146	Informs the compiler that a given compilation unit may reference write-through-enabled or cache-inhibited storage.
"-qhot" on page 153	Performs high-order loop analysis and transformations (HOT) during optimization.

Table 17. Optimization and tuning options (continued)

Option name	Description
"-qignerrno" on page 156	Allows the compiler to perform optimizations as if system calls would not modify errno.
"-qipa" on page 160	Enables or customizes a class of optimizations known as interprocedural analysis (IPA).
"-qisolated_call" on page 165	Specifies functions in the source file that have no side effects other than those implied by their parameters.
"-qlibansi" on page 169	Assumes that all functions with the name of an ANSI C library function are in fact the system functions.
"-qmaxmem" on page 174	Limits the amount of memory that the compiler allocates while performing specific, memory-intensive optimizations to the specified number of kilobytes.
"-qpdf1, -qpdf2" on page 180	Tunes optimizations through *profile-directed feedback* (PDF), where results from sample program execution are used to improve optimization near conditional branches and in frequently executed code sections.
"-qprefetch" on page 184	Inserts prefetch instructions automatically where there are opportunities to improve code performance.
"-qrestrict" on page 191	Specifying this option is equivalent to adding the restrict keyword to the pointer parameters within all functions, except that you do not need to modify the source file.
"-qshowpdf" on page 197	When used with **-qpdf1** and a minimum optimization level of **-O2** at compile and link steps, creates a PDF map file that contains additional profiling information for all procedures in your application.
"-qsimd" on page 198	Controls whether the compiler can automatically take advantage of vector instructions for processors that support them. **Equivalent pragma:** #pragma nosimd
"-qsmallstack" on page 200	Minimizes stack usage where possible. Disables optimizations that increase the size of the stack frame.
"-qsmp" on page 201	Enables parallelization of program code.
"-qstrict" on page 208	Ensures that optimizations that are done by default at the **-O3** and higher optimization levels, and, optionally at **-O2**, do not alter the semantics of a program.
"-qstrict_induction" on page 212	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.

Table 17. Optimization and tuning options (continued)

Option name	Description
"-qunwind" on page 218	Specifies whether the call stack can be unwound by code looking through the saved registers on the stack.

The following options are supported by XL C/C++ for GCC compatibility. For details about these options, see the GNU Compiler Collection online documentation at http://gcc.gnu.org/onlinedocs/.

- --sysroot
- -isysroot
- -isystem

Linking

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

Table 18. Linking options

Option name	Description
"-e" on page 93	When used together with the **-shared (-qmkshrobj)** option , specifies an entry point for a shared object.
"-L" on page 79	At link time, searches the directory path for library files specified by the **-l** option.
"-l" on page 128	Searches for the specified library file. The linker searches for *libkey.so*, and then *libkey.a* if *libkey.so* is not found.
"-qcrt, -nostartfiles (-qnocrt)" on page 144	Specifies whether system startup files are to be linked.
"-qlib, -nodefaultlibs (-qnolib)" on page 167	Specifies whether standard system libraries and XL C/C++ libraries are to be linked.
"-R" on page 85	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.
"-static (-qstaticlink)" on page 222	Controls whether static or shared runtime libraries are linked into an application.
"-X (-W)" on page 88	**-Xlinker** *option* or **-Wl,***option* passes the listed option directly to the linker.

The following options are supported by XL C/C++ for GCC compatibility. For details about these options, see the GNU Compiler Collection online documentation at http://gcc.gnu.org/onlinedocs/.

- -idirafter
- -imacros
- -iprefix

- -iquote
- -iwithprefix
- -pie
- -rdynamic
- -Xlinker

Portability and migration

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

Table 19. Portability and migration options

Option name	Description
"-fpack-struct (-qalign)" on page 103	Specifies the alignment of data objects in storage, which avoids performance problems with misaligned data.
"-qxlcompatmacros" on page 217	Controls definition of the following legacy macros: `__xlC__`, `__xlC_ver__`, `C++` `__IBMCPP__` `C++`, `C` `__IBMC__`, and `__xlc__` `C`.

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 20. Compiler customization options

Option name	Description
"@*file* (-qoptfile)" on page 70	Specifies a response file that contains a list of additional command line options to be used for the compilation. Response files typically have the .rsp suffix.
"-B" on page 72	Specifies substitute path names for XL C/C++ components such as the assembler, C preprocessor, and linker.
"-F" on page 77	Names an alternative configuration file or stanza for the compiler.
"-isystem (-qc_stdinc) (C only)" on page 123	Changes the standard search location for the XL C header files.
"-isystem (-qcpp_stdinc) (C++ only)" on page 124	Changes the standard search location for the XL C++ header files.
"-isystem (-qgcc_c_stdinc) (C only)" on page 125	Changes the standard search location for the GNU C system header files.
"-isystem (-qgcc_cpp_stdinc) (C++ only)" on page 127	Changes the standard search location for the GNU C++ system header files.
"-qasm_as" on page 137	Specifies the path and flags used to invoke the assembler in order to handle assembler code in an `asm` assembly statement.

Table 20. Compiler customization options (continued)

Option name	Description
"-qpath" on page 178	Specifies substitute path names for XL C/C++ components such as the compiler, assembler, linker, and preprocessor.
"-qspill" on page 205	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 228	Applies the prefix specified by the **-B** option to the designated components.
"-X (-W)" on page 88	Passes one or more 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.

> When an option supports the **#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.

> 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 239

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 "Compiler predefined macros" on page 305

Examples

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

-### (-#) (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 it 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:

```
xlc myprogram.c -###
```

Related information
- "-v, -V" on page 229

-+ (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 **-x 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.

You cannot use the **-+** option with the **-qsourcetype** or **-x** option.

Predefined macros

None.

Examples

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

```
 xlc -+ myprogram.cplspls
```

Related information
- "-x (-qsourcetype)" on page 231

--help (-qhelp)

Category

Listings, messages, and compiler information

Pragma equivalent

None.

Purpose

Displays the man page of the compiler.

Syntax

```
►►── --help ──────────────────────────────────────────────────►◄


►►── -q─help ─────────────────────────────────────────────────►◄
```

Usage

If you specify the **--help (-qhelp)** option, regardless of whether you provide input files, the compiler man page is displayed and the compilation stops.

Predefined macros

None.

Related information

- "--version (-qversion)"

--version (-qversion)

Category

Listings, messages, and compiler information

Pragma equivalent

None.

Purpose

Displays the version and release of the compiler being invoked.

Syntax

```
►►── --version ───────────────────────────────────────────────►◄


              ┌─noversion─┐
►►── -q ──────┴─version───┴──────────────────────────────────►◄
                       └─=─verbose─┘
```

Defaults

-qnoversion

--version is not set by default.

Parameters

verbose
> Displays information about the version, release, and level of each compiler component installed.

Usage

When you specify **--version (-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_nameVersion: 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_build_date ID:
component_level_ID
```

where:
component_name
> Specifies an installed component, such as the low-level optimizer.

component_build_date
> Represents the build date of the installed component.

component_level_ID
> Represents the ID associated with the level of the installed component.

For more details, see Example 2.

Predefined macros

None.

Example 1

The output of specifying the **--version (-qversion)** option:

```
IBM XL C/C++ for Linux, V13.1.6 (5765-J08; 5725-C73)
Version: 13.01.0006.0000
```

Example 2

The output of specifying the **-qversion=verbose** option:

```
IBM XL C/C++ for Linux, V13.1.6 (5765-J08; 5725-C73)
Version: 13.01.0006.0000
Driver Version: 13.1.6(C/C++) Level: 150508
ID: _hnbfIvWfEeSjz7qEhQiYJQ
C Front End Version: 15.1.6(Fortran) Level: 150506
ID: _EwaE2-iLEeSbzZ-i2Itj4A
C++ Front End Version: 13.1.6(C/C++) Level: 150511
ID: _YU-wovhCEeSjz7qEhQiYJQ
High-Level Optimizer Version: 13.1.6(C/C++) and 15.1.6(Fortran)
Level: 150512 ID: _mSHAgvkLEeSjz7qEhQiYJQ
Low-Level Optimizer Version: 13.1.6(C/C++) and 15.1.6(Fortran)
Level: 150511 ID: _YY5AQvhCEeSjz7qEhQiYJQ
```

Related information

- "-qsaveopt" on page 195

@*file* (-qoptfile)
Category

Compiler customization

Pragma equivalent

None.

Purpose

Specifies a response file that contains a list of additional command line options to be used for the compilation. Response files typically have the .rsp suffix.

Syntax

```
►►── @──filename──────────────────────────────────────────────►◄
```

```
►►── -q──optfile──=──filename────────────────────────────────►◄
```

Defaults

None.

Parameters

filename
> Specifies the name of the response 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 response file follows these rules:

- Specify the options you want to include in the file with the same syntax as on the command line. The response 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 response 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 @*file* **(-qoptfile)** option from the command line, and sequentially inserts the options included in the file before the other subsequent options that you specify.

The @*file* **(-qoptfile)** option is also valid within a response file. The files that contain another response file are processed in a depth-first manner. The compiler avoids infinite loops by detecting and ignoring cycles in response file inclusion.

If @*file* **(-qoptfile)** and **-qsaveopt** are specified on the same command line, the original command line is used for **-qsaveopt**. A new line for each response file is included representing the contents of each response file. The options contained in the file are saved to the compiled object file.

Predefined macros

None.

Example 1

This is an example of specifying a response file.

```
$ cat options.file
# To perform optimization at -O3 level, and high-order
# loop analysis and transformations during optimization
-O3 -qhot
# To generate position-independent code
-fPIC

$ xlC -qlist @options.file -qipa test.c
```

The preceding example is equivalent to the following invocation:

```
$ xlC -qlist -O3 -qhot -fPIC -qipa test.c
```

Example 2

This is an example of specifying a response file that contains @*file* **(-qoptfile)** with a cycle.

```
$ cat options.file1
# To perform optimization at -O3 level, and high-order
# loop analysis and transformations during optimization
-O3 -qhot
# To include the -qoptfile option in the same response file
@options.file1
# To generate position-independent code
-fPIC
# To produce a compiler listing file
-qlist

$ xlC -qlist @options.file1 -qipa test.c
```

The preceding example is equivalent to the following invocation:

```
$ xlC -qlist -O3 -qhot -fPIC -qlist -qipa test.c
```

Example 3

This is an example of specifying a response file that contains @*file* **(-qoptfile)** without a cycle.

```
$ cat options.file1
-O3 -qhot
@options.file2
-qalias=ansi

$ cat options.file2
-qchars=signed

$ xlC @options.file1 test.c
```

The preceding example is equivalent to the following invocation:

```
$ xlC -O3 -qhot -qalias=ansi -qchars=signed test.c
```

Example 4

This is an example of specifying **-qsaveopt** and @*file* **(-qoptfile)** on the same command line.

```
$ cat options.file3
-O3
-qhot

$ xlC -qsaveopt -qipa @options.file3 test.c -c

$ what test.o
test.o:
opt f xlC -qsaveopt -qipa @options.file3 test.c -c
optfile options.file3 -O3 -qhot
```

Related information
- "-qsaveopt" on page 195

-B

Category

Compiler customization

Pragma equivalent

None.

Purpose

Specifies substitute path names for XL C/C++ components such as the assembler, C preprocessor, 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. However, it is preferred that you use the **-qpath** option to accomplish this instead.

Syntax

```
►►─── -B ──────────────────────────────────────────────────────────────►◄
          └─prefix─┘
```

Defaults

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

Parameters

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

Usage

The **-t** option specifies the programs to which the **-B** prefix name is to be
appended; see "-t" on page 228 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:

```
xlc -tcbI -B/home/jim/opt/ibm/xlC/13.1.6/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 178
- "-t" on page 228
- "Invoking the compiler" on page 1
- The **-B** option that GCC provides. For details, see the GCC online
  documentation at http://gcc.gnu.org/onlinedocs/.

-C, -C!

Category

Output control

Pragma equivalent
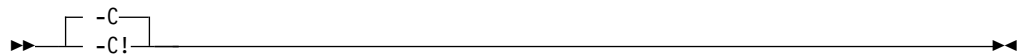
None.

Purpose

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

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

Syntax

```
>>──┬──-C──┬──────────────────────────────────────────────────────────><
    └──-C!─┘
```

Defaults

-C

Usage

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

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

Predefined macros

None.

Examples

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

```
xlc myprogram.c -P -C
```

Related information
- "-E" on page 75
- "-P" on page 84

-D

Category

Language element control

Pragma equivalent

None.

Purpose

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

Syntax

```
►►── -D─name─────────────────────────────────────────────────────►◄
                └─=──definition─┘
```

Defaults

Not applicable.

Parameters

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

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

Usage

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

The **-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:

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

Related information
- "-U" on page 87
- "Compiler predefined macros" on page 305

-E

Category

Output control

Pragma equivalent

None.

Purpose

Preprocesses the source files named in the compiler invocation, without compiling.

Syntax

▶▶── -E──◀◀

Defaults

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

Usage

Source files with unrecognized file name suffixes are treated and preprocessed as C files.

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** and **-fsyntax-only (-qsyntaxonly)** options. The combination of **-E -o** stores the preprocessed result in the file specified by **-o**.

Predefined macros

None.

Examples

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

```
xlc 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,b) ;  /* Comment in a macro function argument*/
```

the output will be:

```
int a ;

int b ;
```

```
int c ;

c = a + b ;
```

Related information
- "-C, -C!" on page 73
- "-P" on page 84
- "-fsyntax-only (-qsyntaxonly)" on page 109

-F

Category

Compiler customization

Pragma equivalent

None.

Purpose

Names an alternative configuration file or stanza for the compiler.

Note: This option is not equivalent to the **-F** option that GCC provides.

Syntax

►►── -F──┬─*file_path*─────────┬──►◄
         │   └─:─*stanza*─┘ │
         └─:─*stanza*────────────┘

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 **xlc**, but you specify the **c99** stanza, the compiler will use all the settings specified in the **c99** stanza.

Usage

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

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

Predefined macros

None.

Examples

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

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

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

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

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

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

Related information
- "Using custom compiler configuration files" on page 38
- "-B" on page 72
- "-t" on page 228
- "-X (-W)" on page 88
- "Specifying compiler options in a configuration file" on page 6
- "Compile-time and link-time environment variables" on page 18

-I

Category

Input control

Pragma equivalent

None.

Purpose

Adds a directory to the search path for include files.

Syntax

►►— -I—*directory_path*————————————————————————————►◄

Defaults

See "Directory search sequence for included files" on page 9 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 **-nostdinc** or **-nostdinc++** (**-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:

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

Related information
- "-qidirfirst" on page 155
- "-qstdinc, -qnostdinc (-nostdinc, -nostdinc++)" on page 206
- "-include (-qinclude)" on page 122
- "Directory search sequence for included files" on page 9
- "Specifying compiler options in a configuration file" on page 6

-L

Category

Linking

Pragma equivalent

None.

Purpose

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

Syntax

►►— -L—*directory_path*————————————————————————►◄

Defaults

The default is to search only the standard directories. See the compiler configuration file for the directories that are set by default.

Parameters

directory_path
   The path for the directory which should be searched for library files.

Usage

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.

The **-L** compiler option is cumulative. Subsequent occurrences of **-L** on the command line do not replace, but add to, any directory paths specified by earlier occurrences of **-L**.

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:

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

Related information
- "-l" on page 128
- "-R" on page 85

-O, -qoptimize
Category

Optimization and tuning

Purpose

Specifies whether to optimize code during compilation and, if so, at which level.

Syntax

```
                         ┌─noopt─────────┐
                         ├─nooptimize────┤
►►──┬──-q─┬─────────┬────┼──optimize─────┼─────────────────────────►◄
    │     └─opt─────┘    └──opt─┘  ┌─=─┬─0─┐
    │                               │   ├─2─┤
    │                               │   ├─3─┤
    │                               │   ├─4─┤
    │                               │   └─5─┘
    ├──-O0───────────────────────────────────┤
    ├──-O────────────────────────────────────┤
    ├──-O2───────────────────────────────────┤
    ├──-O3───────────────────────────────────┤
    ├──-O4───────────────────────────────────┤
    ├──-O5───────────────────────────────────┤
    └──-Ofast─────────────────────────────────┘
```
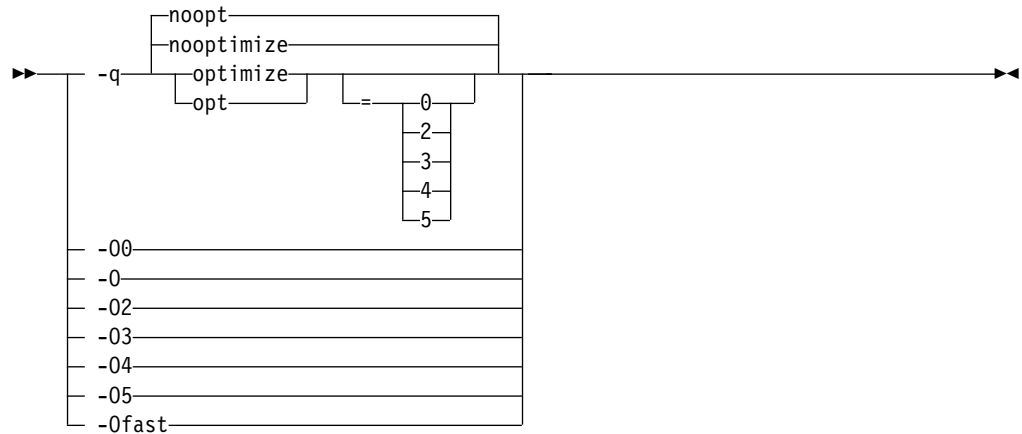
Defaults

-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:

- 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

– It 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 ;
 }
  ...
```

- Both **-O2** and **-O3** conform to the following IEEE rules.

  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.

-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 "-ftrapping-math (-qflttrap)" on page 111 to see the behavior of the compiler when you specify **optimize** options with the **-ftrapping-math (-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 **-mcpu** and **-mtune** 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**, **-mcpu**, and **-mtune** options will override the settings implied by the **-O4** option.

This option follows the "last option wins" conflict resolution rule, so any of the options that are modified by **-O4** can be subsequently changed.

-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**, **-mcpu**, and **-mtune** options will override the settings implied by the **-O5** option.

-Ofast

This option is the same as **-O3 -qhot -D__FAST_MATH__**.

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.

If optimization level **-O3** or higher is specified on the command line, the **-qhot** and **-qipa** options that are set by the optimization level cannot be overridden by `#pragma option_override(`*identifier*`, "opt(level, 0)")` or `#pragma option_override(`*identifier*`, "opt(level, 2)")`.

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:

```
xlc myprogram.c -O3
```

Related information

- "-qhot" on page 153
- "-qipa" on page 160
- "-qpdf1, -qpdf2" on page 180
- "-qstrict" on page 208
- "Optimizing your applications" in the *XL C/C++ Optimization and Programming Guide*.
- "#pragma option_override" on page 244

-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 but with a .i suffix.

Note: This option is not equivalent to the GCC option **-P** .

Syntax

▶▶── -P ───▶◀

Defaults

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

Usage

Source files with unrecognized file name suffixes are preprocessed as C files except those with a .i suffix.

`#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 **-fsyntax-only (-qsyntaxonly)** option.

Predefined macros

None.

Related information
- "-C, -C!" on page 73
- "-E" on page 75
- "-fsyntax-only (-qsyntaxonly)" on page 109

-R

Category

Linking

Pragma equivalent

None.

Purpose

At link time, writes search paths for shared libraries into the executable, so that these directories are searched at program run time for any required shared libraries.

Syntax

►►── -R──*directory_path*──►◄

Defaults

The default is to include only the standard directories. See the compiler configuration file for the directories that are set by default.

Usage

If the **-R***directory_path* option is specified both in the configuration file and on the command line, the paths specified in the configuration file are searched first at run time.

The **-R** compiler option is cumulative. Subsequent occurrences of **-R** on the command line do not replace, but add to, any directory paths specified by earlier occurrences of **-R**.

Predefined macros

None.

Examples

To compile myprogram.c so that the directory /usr/tmp/old is searched at run time along with standard directories for the dynamic library libspfiles.so, enter:

```
xlc myprogram.c -lspfiles -R/usr/tmp/old
```

Related information
- "-L" on page 79

-S

Category

Output control

Pragma equivalent

None.

Purpose

Generates an assembler language file for each source file.

The resulting file has a .s suffix and can be assembled to produce object .o files or an executable file (a.out).

Syntax

►►— -S——————————————————————————————►◄

Defaults

Not applicable.

Usage

You can invoke the assembler with any compiler invocation command. For example,

```
xlc 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:

```
xlc myprogram1.c myprogram2.c -o -S
```

Predefined macros

None.

Examples

To compile myprogram.c to produce an assembler language file myprogram.s, enter:

```
xlc myprogram.c -S
```

To assemble this program to produce an object file myprogram.o, enter:

```
xlc myprogram.s -c
```

To compile `myprogram.c` to produce an assembler language file `asmprogram.s`, enter:

```
xlc myprogram.c -S -o asmprogram.s
```

Related information

- "-E" on page 75
- "-P" on page 84

-U

Category

Language element control

Pragma equivalent

None.

Purpose

Undefines a macro defined by the compiler or by the **-D** compiler option.

Syntax

```
►►── -U─name────────────────────────────────────────►◄
```

Defaults

Many macros are predefined by the compiler; see "Compiler predefined macros" on page 305 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; see the configuration file for your system for more information.

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:

```
xlc myprogram.c -U__unix
```

Related information

- "-D" on page 74

-X (-W)

Category

Compiler customization

Pragma equivalent

None.

Purpose

Passes one or more options to a component that is executed during compilation.

Syntax

```
                                               (1)
►►──-X──┬─assembler────┬──────────option───────────────────────────────►◄
        ├─preprocessor─┤
        ├─linker───────┤
        ├─partitioner──┤
        ├─wc2llvm──────┤
        ├─llvm2ptx─────┤
        ├─ptxas────────┤
        └─nvcc─────────┘
```
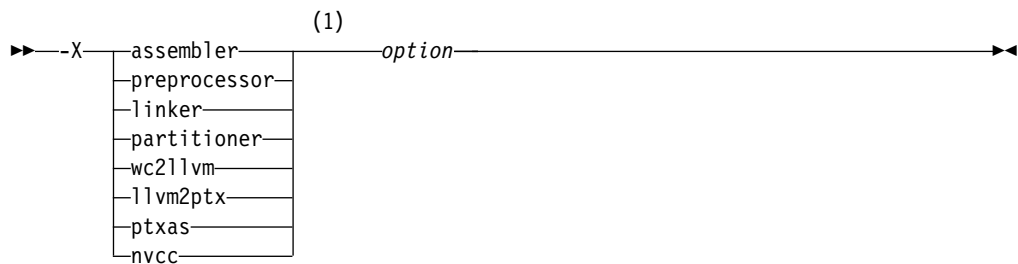
Notes:

1   You must insert at least one space before *option*.

```
       ┌────────┐  ┌──────────┐
►►── -W─┼──@──┼──┼──,──option──┼────────────────────────────────────────►◄
        ├─a─┤
        ├─b─┤
        ├─c─┤
        ├─C─┤
        ├─d─┤
        ├─I─┤
        ├─L─┤
        ├─l─┤
        ├─n─┤
        ├─p─┤
        ├─s─┤
        ├─w─┤
        └─x─┘
```
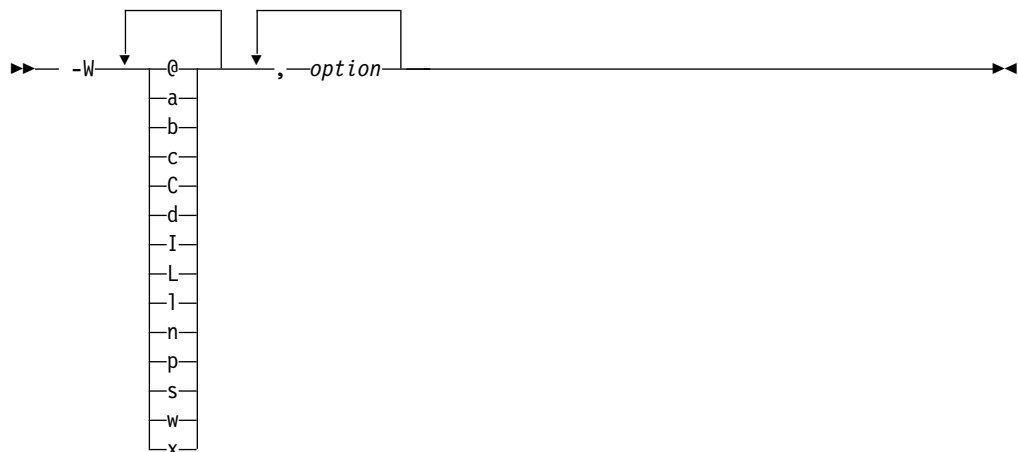
Parameters

option
    Any option that is valid for the component to which it is being passed.

        Notes: ▶ GPU

- You can find the NVVM-IR to PTX translator options in the libNVVM API section in the CUDA Toolkit documentation at http://docs.nvidia.com/cuda/libnvvm-api/group__compilation.html under nvvmCompileProgram.
- You can get a list of the PTX assembler options by running **ptxas** from the CUDA Toolkit with **-h**.

`  GPU  ◄`

For **-X**, for details about the options for linking and assembling, see the GNU Compiler Collection online documentation at http://gcc.gnu.org/onlinedocs/.

The following table shows the correspondence between **-X** or **-W** parameters and the component names:

Parameter of -W	Parameter of -X	Description	Component name
► GPU @	ptxas	The PTX assembler	ptxas
a	assembler	The assembler	as
b		The low-level optimizer	xlCcode
c, C		The C and C++ compiler front end	xlCentry
d		The disassembler	dis
I (uppercase i)		The high-level optimizer, compile step	ipa
L		The high-level optimizer, link step	ipa
l (lowercase L)	linker	The linker	ld
► GPU n	nvcc	The NVIDIA C compiler, which is used as a device linker	nvcc
p	preprocessor	The preprocessor	xlCentry
► GPU s	partitioner	The XL intermediate language (W-Code) splitter	partitioner
► GPU w	wc2llvm	The XL intermediate language (W-Code) to NVVM-IR translator	wc2llvm
► GPU x	llvm2ptx	The NVVM-IR to PTX translator	llvm2ptx

Usage

In the string following the **-W** option, use a comma as the separator for each option, and do not include any spaces. For the **-X** option, one space is needed before the *option*. 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 **-X** or **-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 **–X** or **–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 **–X** or **–W**.

> **GPU** To use **–W@**, **–Wn**, **–Ws**, **–Ww**, **–Wx**, or their respective **–X** equivalents, you must specify the **–qoffload** option. **GPU** ◀

Predefined macros

None.

Examples

To compile the file `file.c` and pass the linker option **–symbolic** to the linker, enter the following command:

```
xlc -Xlinker -symbolic 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 **–alh**, and the object files are linked with the option **–s** (write list of object files and strip final executable file), issue either of the following commands:

```
xlc -Xassembler -alh produces_warnings.s -Xlinker -s uses_many_symbols.c
```

```
xlc -Wa,-alh produces_warnings.s -Wl,-s uses_many_symbols.c
```

Related information
- "Invoking the compiler" on page 1

-Werror (-qhalt)
Category

Error checking and debugging

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

```
▶▶──-Werror─────────────────────────────────────────────▶◀


▶▶───-qhalt──=w──────────────────────────────────────────▶◀
```

Defaults

By default, **-Werror (-qhalt=w)** is disabled.

Parameters

w   Specifies that compilation is to stop for warnings (W) and all types of errors.

Predefined macros

None.

Examples

To compile `myprogram.c` so that compilation stops if a warning or higher level message occurs, enter:

```
xlc myprogram.c -Werror
```

-Wunsupported-xl-macro
Category

Error checking and debugging

Pragma equivalent

None.

Purpose

Checks whether any unsupported XL macro is used.

Syntax

►►──-Wunsupported-xl-macro──►◄

Defaults

By default, **-Wunsupported-xl-macro** is disabled.

Usage

Some macros that might be supported by other XL compilers are unsupported in IBM XL C/C++ for Linux, V13.1.6.

You can specify the **-Wunsupported-xl-macro** option to check whether any unsupported macro is used. If an unsupported macro is used, the compiler issues a warning message.

Predefined macros

None.

Related information
"Unsupported macros from other XL compilers" on page 312
"-qxlcompatmacros" on page 217

-c
Category

Output control

Pragma equivalent

None.

Purpose

Instructs the compiler to compile or assemble the source files only but do not link. With this option, the output is a .o file for each source file.

Syntax

►►── -c──►◄

Defaults

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

Usage

When this option is in effect, the compiler creates an output object file, *file_name*.o, for each valid source file, such as *file_name*.c, *file_name*.i, *file_name*.C, *file_name*.cpp, or *file_name*.s. 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 **-fsyntax-only (-qsyntaxonly)** option is specified.

Predefined macros

None.

Examples

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

```
xlc myprogram.c -c
```

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

```
xlc myprogram.c -c -o new.o
```

Related information
- "-E" on page 75
- "-o" on page 134
- "-P" on page 84
- "-fsyntax-only (-qsyntaxonly)" on page 109

-dM (-qshowmacros)
Category

"Output control" on page 51

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

```
►►─── -dM ──────────────────────────────────────────────────────────────►◄
```

```
                 ┌─noshowmacros─┐
►►─── -q ─────────┴─showmacros───┴───────────────────────────────────────►◄
```

Defaults

-qnoshowmacros

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 75
- "-P" on page 84

-e

Category

Linking

Pragma equivalent

None.

Purpose

Specifies an entry point for a shared object when used together with the **-shared (-qmkshrobj)** option.

Syntax

```
►►─── -e ────entry_name────────────────────────────────────────────────────◄◄
```

Defaults

None.

Parameters

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

Usage

Specify the **-e** option only with the **-shared (-qmkshrobj)** option.

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
• "-shared (-qmkshrobj)" on page 220

-fasm (-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 take effect.

Syntax

```
                  ┌─asm────┐
►►─── -f──┴─no-asm─┴──────────────────────────────────────────────────────◄◄
```

```
        ┌─asm─┐
        │     ┌─gcc─┐
        │   ┌─┴─────┴─┐
        │ ┌─┘ =       └─
►►─ -q─┬─┴─────────────┬──────────────────────────────►◄
       └─noasm─────────┘
```

Defaults

-qasm=gcc or -fasm

Parameters

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

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

Usage

▶ **C**   At language levels **stdc89** and **stdc99**, token asm is not a keyword. At all the other language levels, token asm is treated as a keyword. **C** ◀

▶ **C++**   The tokens asm, __asm, and __asm__ are keywords at all language levels. **C++** ◀

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

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" on page 137
- "-std (-qlanglvl)" on page 224
- "Inline assembly statements" in the *XL C/C++ Language Reference*

-fcommon (-qcommon)
Category

Object code control

Pragma equivalent

None.

Purpose

Controls where uninitialized global variables are allocated.

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

Syntax

```
►►── -f──┬─common────┬──────────────────────────────────────────►◄
         └─no-common─┘
```

```
►►── -q──┬─common───┬──────────────────────────────────────────────►◄
         └─nocommon─┘
```
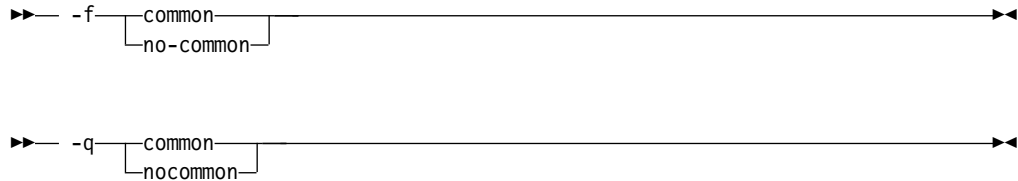
Defaults

- ▸ C ◂ **-fcommon (-qcommon)** except when **-shared (-qmkshrobj)** is specified; **-fno-common (-qnocommon)** when **-shared (-qmkshrobj)** is specified.
- ▸ C++ ◂ **-fno-common (-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 **-fcommon (-qcommon)** produces the equivalent of the following assembly code:
```
.comm _a,4
.comm _b,4
```

Compiling with **-fno-common (-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
- "-shared (-qmkshrobj)" on page 220
- "The common and nocommon variable attribute" in the *XL C/C++ Language Reference*

- "The section variable attribute" in the *XL C/C++ Language Reference*

-fdollars-in-identifiers (-qdollar)
Category

Language element control

Pragma equivalent

None

Purpose

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

When **-fdollars-in-identifiers** or **-qdollar** is in effect, the dollar symbol $ in an identifier is treated as a base character.

Syntax

```
                  ┌─dollars-in-identifiers────┐
►►──-f────┴─no-dollars-in-identifiers─┘──────────────────────────────►◄
```

```
            ┌─dollar──┐
►►── -q────┴─nodollar─┘────────────────────────────────────────────►◄
```
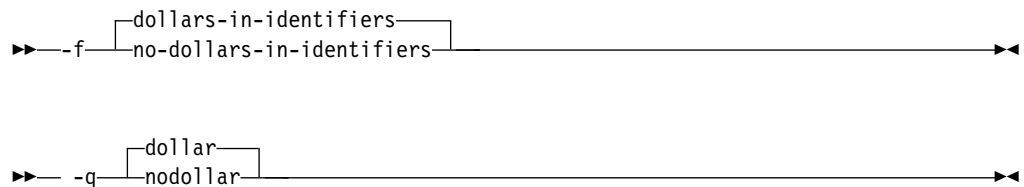
Defaults

-fdollars-in-identifiers or **-qdollar**

Predefined macros

None.

Examples

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

Related information
- "-std (-qlanglvl)" on page 224

-fdump-class-hierarchy (-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

```
►►── -f─dump-class-hierarchy─────────────────────────────────────◄◄
```

```
►►── -q─dump_class_hierarchy─────────────────────────────────────◄◄
```

Defaults

Not applicable.

Usage

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

Predefined macros

None.

Examples

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

```
xlc++ myprogram.C -fdump-class-hierarchy
```

-fexceptions (-qeh) (C++ only)
Category

Object code control

Pragma equivalent

None.

Purpose

The **-fexceptions** option controls whether exception handling is allowed in the module being compiled. The **-qeh** option controls whether exception handling is enabled in the module being compiled.

Syntax

```
                  ┌─exceptions──┐
►►── -f───┴─no-exceptions─┴──────────────────────────────────────◄◄
```

```
              ┌─eh──┐
►►── -q───┴─noeh─┴───────────────────────────────────────────────◄◄
```

Defaults

-fexceptions or **-qeh**

Usage

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.

The difference between **-fexceptions** and **-qeh** is that **-fexceptions** allows exception handling while **-qeh** informs the compiler that exceptions need to be handled. When exceptions are not allowed during a particular compilation process, the compiler assumes that exceptions do not need to be handled; exceptions might originate from code that is compiled separately and still occur.

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 **-fexceptions (-qeh)** is in effect; otherwise, it is undefined.

Related information
- "-qrtti, -fno-rtti (-qnortti) (C++ only)" on page 194

-finline-functions (-qinline)
Category

Optimization and tuning

Pragma equivalent

None.

Purpose

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

Syntax

►►──-finline-functions──►◄

```
        ┌─-qnoinline─────────────────────────────────────────┐
►►──┬────┴─────────────┬───────────────────────────────────────►◄
    └─-qinline─┬────────────────────────────────────┬─┘
              │        ┌─ : ─────────┐               │
              │    ┌───▼──auto───────┤               │
              ├─=──┼──────noauto─────┤               │
              │    └──────level──=──number─┘         │
              │        ┌─ : ─────────┐               │
              └─┬─+─┬──▼──function_name──┘
                └─-─┘
```
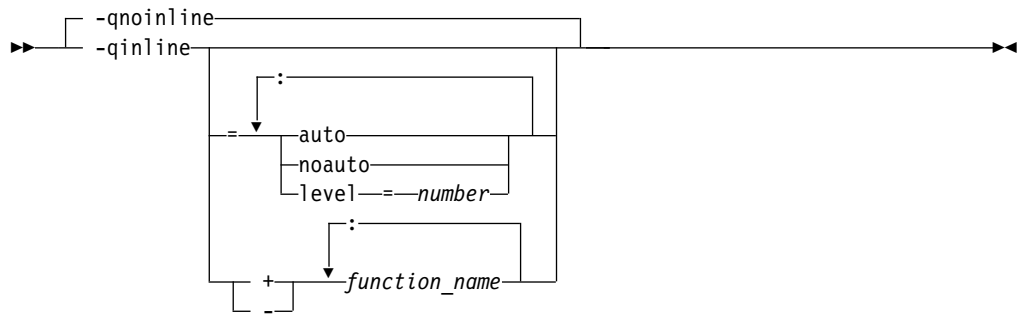
Defaults

If **-qinline** is not specified, the default option is as follows:

- **-qnoinline** at the **-O0** or **-qnoopt** optimization level
- **-qinline=noauto:level=5** at the **-O2** optimization level
- **-qinline=auto:level=5** at the **-O2 -qipa**, **-O3** or higher optimization level

If **-qinline** is specified without any suboptions, the default option is
-qinline=auto:level=5.

Parameters

auto | noauto

Enables or disables automatic inlining. When option **-qinline=auto** is in effect, all functions are considered for inlining by the compiler. When option **-qinline=noauto** is in effect, only the following types of functions are considered for inlining:

- Functions that are defined with the inline specifier
- Small functions that are identified by the compiler

The compiler determines whether a function is appropriate for inlining, and enabling automatic inlining does not guarantee that a function is inlined.

level=*number*

Indicates the relative degree of inlining. The values for *number* must be integers in the range 0 - 10 inclusive. The default value for *number* is 5. The greater the value of *number*, the more aggressive inlining the compiler conducts.

function_name

If *function_name* is specified after the **-qinline+** option, the named function must be inlined. If *function_name* is specified after the **-qinline-** option, the named function must not be inlined. ► C++ The *function_name* must be the mangled name of the function. You can find the mangled function name in the listing file. C++ ◄

Usage

You can specify ► C++ **-qinline** C++ ◄ or specify **-qinline** with any optimization level of ► C++ **-O** C++ ◄ , **-O2**, **-O3**, **-O4**, or **-O5** to enable inlining of functions, including those functions that are declared with the inline specifier ► C++ or that are defined within a class declaration C++ ◄ .

When **-qinline** is in effect, the compiler determines whether inlining a specific function can improve performance. That is, whether a function is appropriate for inlining is subject to two factors: limits on the number of inlined calls and the

amount of code size increase as a result. Therefore, enabling inlining a function does not guarantee that function will be inlined.

Because inlining does not always improve runtime performance, you need to test the effects of this option on your code. Do not attempt to inline recursive or mutually recursive functions.

You can use the **-qinline+<**function_name**>** or **-qinline-<**function_name**>** option to specify the functions that must be inlined or must not be inlined.

 ▶ IBM The **-qinline-<**function_name**>** option takes higher precedence than the always_inline or __always_inline__ attribute. When you specify both the always_inline or __always_inline__ attribute and the **-qinline-<**function_name**>** option to a function, that function is not inlined. IBM ◀

Specifying **-qnoinline** disables all inlining, including that achieved by the high-level optimizer with the **-qipa** option, and functions declared explicitly as inline. However, the **-qnoinline** option does not affect the inlining of the following functions:

- ▶ IBM Functions that are specified with the always_inline or __always_inline__ attribute IBM ◀
- Functions that are specified with the **-qinline+<**function_name**>** option

If you specify the **-g** option to generate debugging information, the inlining effect of **-qinline** might be suppressed.

If you specify the **-qcompact** option to avoid optimizations that increase code size, the inlining effect of **-qinline** might be suppressed.

Predefined macros

None.

Examples

Example 1

To compile myprogram.c so that no functions are inlined, use the following command:

```
xlc myprogram.c -O2 -qnoinline
```

However, if some functions in myprogram.c are specified with ▶ IBM the always_inline or __always_inline__ attribute IBM ◀ , the **-qnoinline** option has no effect on these functions and they are still inlined.

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

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

You can specify an inlining level 6 - 10 to achieve 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 of 7, you enter:

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

Example 2

Assuming `myprogram.c` contains the `salary`, `taxes`, `expenses`, and `benefits` functions, you can use the following command to compile `myprogram.c` to inline these functions:

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

If you do not want the functions `salary`, `taxes`, `expenses`, and `benefits` to be inlined, use the following command to compile `myprogram.c`:

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

You can also disable automatic inlining and specify certain functions to be inlined with the **-qinline+** option. Consider the following example:

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

In this case, the functions `salary`, `taxes`, and `benefits` are inlined. Functions that are specified with ▶ **IBM** the `always_inline` or `__always_inline__` attribute **IBM** ◀ or declared with the `inline` specifier are also inlined. No other functions are inlined.

You cannot mix the **+** and **-** suboptions with each other or with other **-qinline** suboptions. For example, the following options are invalid suboption combinations:

```
-qinline+increase-decrease    // Invalid
-qinline=level=5+increase     // Invalid
```

However, you can use multiple **-qinline** options separately. See the following example:

```
-qinline+increase -qinline-decrease -qinline=noauto:level=5
```

▶ **C++** In C++, you can use the **-qinline+** and **-qinline-** options in the same way as in example 2; however, you must specify the mangled function names instead of the actual function names after these options. **C++** ◀

Related information
- "-g" on page 120
- "-qipa" on page 160
- "-O, -qoptimize" on page 80
- "Compiler listings" on page 13
- "always_inline (IBM extension)" in the *XL C/C++ Language Reference*

-fPIC (-qpic)
Category

Object code control

Pragma equivalent

None.

Purpose

Generates position-independent code required for use in shared libraries.

Syntax

```
>>-- -f---┬-no-PIC-┬------------------------------------------------><
          └-PIC----┘
```

```
>>-- -q---┬-nopic-┬--------------------------------------------------><
          └-pic---┘
```

Defaults
- **-fno-PIC**, or **-qnopic**

Usage

When **-fPIC (-qpic)** is in effect, the compiler generates position-independent code.

If a thread local storage (TLS) model is not specified, the position-independent code setting determines the default TLS model:
- When **-fno-PIC (-qnopic)** is in effect, the default TLS model is `local-exec`.
- When **-fPIC (-qpic)** is in effect, the default TLS model is `general-dynamic`.

If the `initial-exec` TLS model is in effect, different code sequences are used depending on different position-independent code settings.

You must compile all the compilation units that are not part of a shared library with **-fno-PIC (-qnopic)** and that are part of a shared library with **-fPIC (-qpic)**.

Predefined macros

None.

Examples

To compile a shared library `libmylib.so`, use the following commands:

```
xlc mylib.c -fPIC -c -o mylib.o
xlc -shared mylib -o libmylib.so.1
```

Related information
- "-shared (-qmkshrobj)" on page 220

-fpack-struct (-qalign)
Category

Portability and migration

Purpose

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

Syntax

```
►►──-fpack-struct──────────────────────────────────────────►◄
```

```
►►── -q─align──┬─=linuxppc──┬──────────────────────────────►◄
               └─=bit_packed─┘
```

Defaults

-qalign=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.

Usage

If you use the **-fpack-struct (-qalign=bit_packed)** or **-qalign=linuxppc** option more than once on the command line, the last alignment rule specified applies to the file.

Note: When using **-fpack-struct (-qalign=bit_packed)** or **-qalign=linuxppc** , all system headers are also compiled with **-fpack-struct (-qalign=bit_packed)** or **-qalign=linuxppc** . For a complete explanation of the option as well as usage considerations, see "Aligning data" in the *XL C/C++ Optimization and Programming Guide*.

Predefined macros

None.

Related information
- "Supported GCC pragmas" on page 240
- "Aligning data" in the *XL C/C++ Optimization and Programming Guide*
- "The aligned variable attribute" in the *XL C/C++ Language Reference*
- "The packed variable attribute" in the *XL C/C++ Language Reference*

-fstack-protector (-qstackprotect)
Category

Error checking and debugging

Pragma equivalent

None.

Purpose

Provides protection against malicious input data or programming errors that overwrite or corrupt the stack.

Syntax

```
                    ┌─ -fno-stack-protector ─────────────────────┐
►►─┬─────────────────┼─ -fstack-protector ──────────────────────┼───────────────────────►◄
                     ├─ -fstack-protector-all ───────────────────┤
                     ├─ -fstack-protector-strong ────────────────┤
                     └─ -fstack-protector --parm ssp-buffer-size──=──N─┘


                    ┌─ nostackprotect ──────────┐
                    │                      ┌─ proc ──┐
►►─── -q ──┴─ stackprotect ────=──┼─ all ───┼────────────────────────────────────────►◄
                                   ├─ size ──=──N ─┤
                                   ├─ strong ─┤
                                   └─ warn ──┘
```
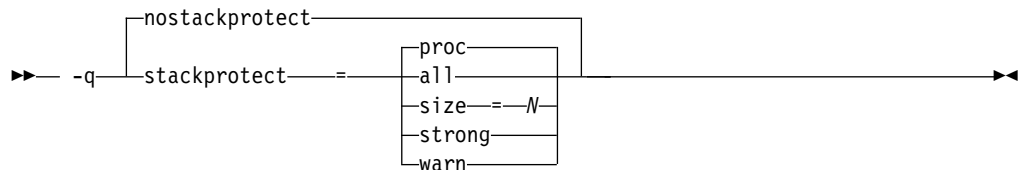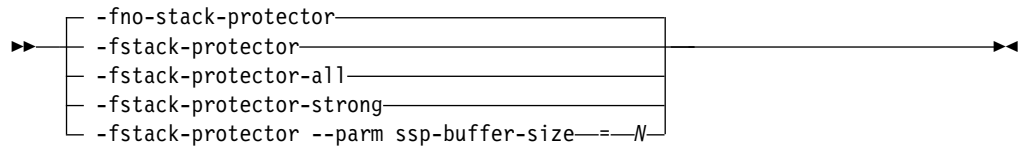
Defaults

-fno-stack-protector (-qnostackprotect) when -fstack-protector (-qstackprotect) is not specified.

-fstack-protector (-qstackprotect=proc) when -fstack-protector (-qstackprotect) is specified without a suboption.

Parameters

`all`
Protects all functions whether or not functions have vulnerable objects.

`proc (-qstackprotect only)`
Provides code to prevent buffer overflows. It is equivalent to the **`-fstack-protector`** option.

`size=` N **` (-qstackprotect only), --parm ssp-buffer-size=`** N **` (-fstack-protector only)`**
Protects all functions that contain automatic arrays whose sizes are greater than or equal to N bytes. The default size is 8 bytes when the **`-fstack-protector (-qstackprotect)`** option is enabled.

`strong`
Protects additional functions that have local array definitions or that have references to local frame addresses.

`warn (-qstackprotect only)`
Issues warnings when the size of the array contained in the function is less than N bytes. It is equivalent to the **`-Wstack-protector`** option. The **`-Wstack-protector`** option is active only when **`-fstack-protector`** is active. For more details about the **`-Wstack-protector`** option, see the GNU Compiler Collection online documentation at http://gcc.gnu.org/onlinedocs/.

Usage

-fstack-protector (-qstackprotect) generates extra code to protect functions with vulnerable objects against stack corruption. The **-fstack-protector (-qstackprotect)** option is disabled by default because it can degrade runtime performance.

To generate code to protect all functions, enter the following command:
```
xlc myprogram.c -fstack-protector=all
```

To generate code to protect functions with objects of certain size, enter the following command with the **size=** parameter set to the object size indicated in bytes:
```
xlc myprogram.c -qstackprotect=size=8
```

Predefined macros

None.

-fsigned-char, -funsigned-char (-qchars)
Category

Floating-point and integer control

Pragma equivalent

None.

Purpose

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

Syntax

```
►►── -f──┬─unsigned─┬──char──────────────────────────────────────────►◄
         ├─signed───┤
         ├─no-unsigned┤
         └─no-signed──┘


►►── -q──chars──=──┬─unsigned─┬────────────────────────────────────────►◄
                   └─signed───┘
```

Defaults

-funsigned-char or -qchars=unsigned

Parameters

unsigned
> Variables of type char are treated as unsigned char.
>
> **-fno-signed-char** is equivalent to **-funsigned-char**.

signed
> Variables of type char are treated as signed char.

-fno-unsigned-char is equivalent to **-fsigned-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.

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.

-fstandalone-debug

Category

Error checking and debugging

Pragma equivalent

None.

Purpose

When used with the **-g** option, controls whether to generate the debugging information for all symbols.

Syntax

```
        ┌─-fno-standalone-debug─┐
►►──────┴─-fstandalone-debug────┴─────────────────────────────────►◄
```

Defaults

-fno-standalone-debug

Usage

This option takes effect only when it is specified with the **-g** option; otherwise, it is ignored.

When **-fstandalone-debug** is in effect, the compiler generates the debugging information for all symbols whether or not these symbols are referenced by the program. Generating the debugging information for all symbols might increase the size of the object file.

To reduce the size of the object file, you can specify the **-fno-standalone-debug** option to generate debugging information only for symbols that are referenced by the program.

Predefined macros

None.

Related information
- "-g" on page 120

-fstrict-aliasing (-qalias=ansi), -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

```
                              ┌─:─────────────┐
                              │  ┌─restrict────┐
                              ▼  ├─ansi────────┤
                              │  ├─noaddrtaken─┤
►►── -q─alias──=──┴──────────┼─addrtaken───┤──────────────────────────►◄
                                 ├─noansi──────┤
                                 └─norestrict──┘
```
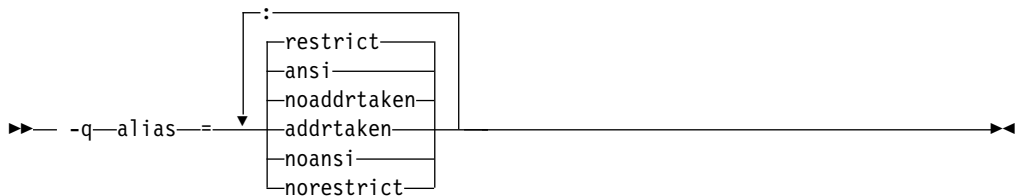
For details about the **-fstrict-aliasing** option, see the GCC information, which is available at http://gcc.gnu.org/onlinedocs/.

Defaults

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

Parameters

addrtaken | noaddrtaken
When **addrtaken** is in effect, the reference of any variable whose address is taken may alias to any pointer type. Any class of variable for which an address has *not* been recorded in the compilation unit is considered disjoint from indirect access through pointers.

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

ansi | noansi
This suboption has no effect unless you also specify an optimization option. You can specify the may_alias attribute for a type that is not subject to type-based aliasing rules.

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.

<pre>restrict | norestrict</pre>
> 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 usually results 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.

Usage

-qalias makes assertions to the compiler about the code that is being compiled. If
the assertions about the code are false, the code that is generated by the compiler
might result in unpredictable behavior 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 that are qualified as `volatile` or `const`. For example, a pointer to a `const int` can point to an `int`.
- ▶ C++ Base type pointers can point to the derived types of that type. C++ ◀

Predefined macros

None.

Examples

To specify worst-case aliasing assumptions when you compile `myprogram.c`, enter:
```
xlc myprogram.c -O -qalias=noansi
```

Related information
- "-qipa" on page 160
- *The may_alias type attribute (IBM extension)* in the *XL C/C++ Language Reference*
- "-qrestrict" on page 191

-fsyntax-only (-qsyntaxonly)
Category

Error checking and debugging

Pragma equivalent

None.

Purpose

Performs syntax checking without generating an object file.

Syntax

```
►►— -f—syntax-only—————————————————————————————◄◄
```

```
►►— -q—syntaxonly—————————————————————————————◄◄
```

Defaults

By default, source files are compiled and linked to generate an executable file.

Usage

The **-P**, **-E**, and **-C** options override the **-fsyntax-only (-qsyntaxonly)** option, which in turn overrides the **-c** and **-o** options.

The **-fsyntax-only (-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:

```
xlc myprogram.c -fsyntax-only
```

Related information
- "-C, -C!" on page 73
- "-c" on page 91
- "-E" on page 75
- "-o" on page 134
- "-P" on page 84

-ftemplate-depth (-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

```
►►— -f—-template-depth—=—number—————————————————◄◄
```

```
►►── -q─templatedepth──=─number───────────────────────────────────────────────►◄
```

Defaults

-ftemplate-depth=256 or **-qtemplatedepth=256**

Parameters

number
> The maximum number of recursive template instantiations. The number can be a value in the range of 1 to 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 256 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:

```
xlc++ myprogram.cpp -ftemplate-depth=400
```

Related information
- "Using C++ templates" in the *XL C/C++ Optimization and Programming Guide*.

-ftrapping-math (-qflttrap)
Category

Error checking and debugging

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

```
►►── -f──┬─notrapping-math─┬──────────────────────────────────────────────►◄
         └─trapping-math───┘
```

```
►►── -q──┬─noflttrap─┬──────────────────────────────────────────────►◄
         └─flttrap───┬────────────────────────────────────┐
                     │           ┌─:─┐                     │
                     │          ┌─zero──────┐              │
                     │          ├─zerodivide┤              │
                     │          ├─und───────┤              │
                     │          ├─underflow─┤              │
                     │          ├─ov────────┤              │
                     │          ├─overflow──┤              │
                     │          ├─inv───────┤              │
                     │          ├─invalid───┤              │
                     │          ├─inex──────┤              │
                     │          ├─inexact───┤              │
                     └─=─────────┼─enable────┤──────────────┘
                                 ├─en────────┤
                                 └─nanq──────┘
```
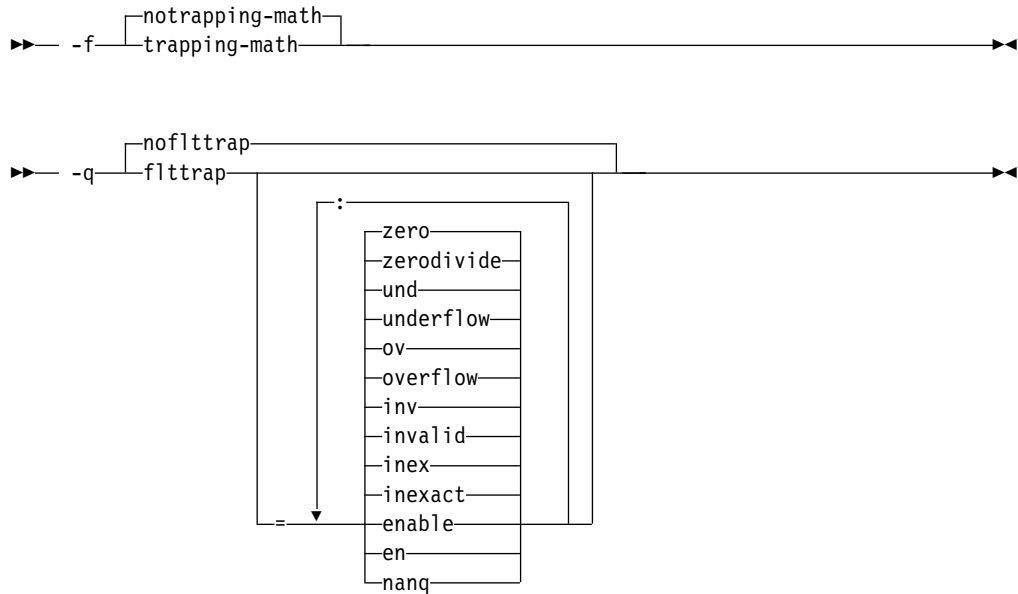
Defaults

-fnotrapping-math or **-qnoflttrap**

Specifying **-qflttrap** option with no suboptions is equivalent to **-qflttrap=overflow:underflow:zerodivide:invalid:inexact**

Parameters

Note: You can specify the following suboptions with **-qflttrap** only.

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.

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

Exceptions will be detected by the hardware, but trapping is not enabled.

It is recommended that you use the **enable** suboption whenever compiling the `main` program with **-ftrapping-math (-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 **-ftrapping-math (-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 **-ftrapping-math (-qflttrap)** to trap any exceptions.

The compiler exhibits behavior as illustrated in the following examples when the **-ftrapping-math (-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.

```
xlc -ftrapping-math 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 147
- "-mcpu (-qarch)" on page 131

-ftls-model (-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 thread-local 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 multithreaded 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

```
                    ┌─tls-model──┬─=global-dynamic─┐
                    │            ├─=local-dynamic──┤
                    │            ├─=initial-exec───┤
                    │            └─=local-exec─────┤
►►── -f──┴─no-tls-model─────────────────────────────┴──────────►◄
```

```
                          ┌──=default───────┐
                 ┌─tls────┼──=global-dynamic─┤
                 │        ├──=initial-exec────┤
                 │        ├──=local-exec──────┤
                 │        └──=local-dynamic───┘
  ►►─── -q───────┴─notls──────────────────────────────────────────────────►◄
```
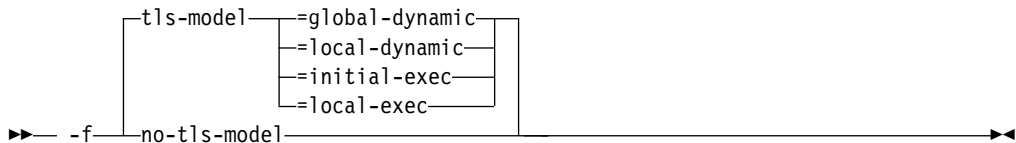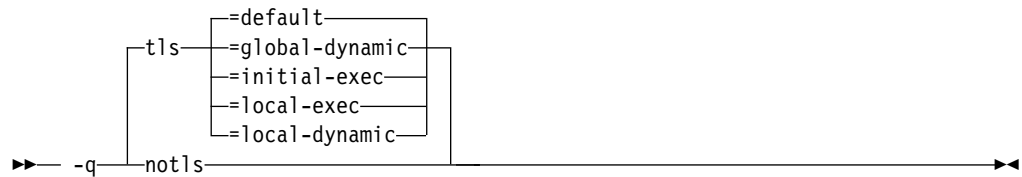
Defaults

-qtls=default

Specifying **-qtls** with no suboption is equivalent to specifying **-qtls=default**.

The default setting for **-ftls-model** is the same as the default setting for **-qtls**.

Parameters

default (-qtls only)
Uses the appropriate model depending on the setting of the **-fPIC (-qpic)** option, which determines whether position-independent code is generated or not. When **-fPIC (-qpic)** is in effect, this suboption results in **-qtls=global-dynamic**. When **-fno-pic (-fno-PIC, -qnopic)** is in effect, this suboption results in **-qtls=initial-exec** .

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.

Predefined macros

None.

Related information
- "-fPIC (-qpic)" on page 102
- "The __thread storage class specifier" in the *XL C/C++ Language Reference*

-ftime-report (-qphsinfo)
Category

Listings, messages, and compiler information

Pragma equivalent

None.

Purpose

Reports the time taken in each compilation phase to standard output.

Syntax

```
►►──-ftime-report──────────────────────────────────────────────►◄


             ┌─nophsinfo─┐
►►── -q──────┴─phsinfo───┴──────────────────────────────────────►◄
```

Defaults

-ftime-report is not on by default.

-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 real time (wall clock time).

The time reported by -q*phsinfo* is in seconds.

Predefined macros

None.

Example

To compile `myprogram.c` and report the time taken for each phase of the compilation, enter the following command:

```
xlc myprogram.c -ftime-report
```

The output looks like:

```
---User Time--- --System Time-- --User+System-- ---Wall Time--- --- Name ---
0.0007 (100.0%) 0.0007 (100.0%) 0.0014 (100.0%) 0.0014 (100.0%) Clang front-end timer
0.0007 (100.0%) 0.0007 (100.0%) 0.0014 (100.0%) 0.0014 (100.0%) Total

Front End - Phase Ends; 0.000/ 0.000
Compilation Time = 0:0.001088
Gen IL Time = 0:0.000288
Optimization Time = 0:0.000264
Code Gen Time = 0:0.000528
```

-funroll-loops (-qunroll), -funroll-all-loops (-qunroll=yes)
Category

Optimization and tuning

Pragma equivalent

#pragma unroll

Purpose

Controls loop unrolling, for improved performance.

-funroll-loops
> Instructs the compiler to perform basic loop unrolling.

-funroll-all-loops
> Instructs the compiler to search for more opportunities for loop unrolling than that performed with **-funroll-loops**. In general, **-funroll-all-loops** has more chances to increase compile time or program size than **-funroll-loops** processing, but it might also improve your application's performance.

When **-funroll-loops** or **-funroll-all-loops** is in effect, the optimizer determines and applies the best unrolling factor for each loop; in some cases, the loop control might be modified to avoid unnecessary branching. The compiler remains the final arbiter of whether the loop is unrolled.

Syntax

Option syntax

```
          ┌─-funroll-loops──┐
►►────┴──-funroll-all-loops──┘────────────────────────────────►◄
```

Option syntax

```
                              ┌─auto─┐
              ┌──unroll──=──┬─yes─┬──┐
              │             ├─no──┤  │
              │             └─n───┘  │
►►─── -q────┴───nounroll──────────────┴──────────────────►◄
```

Defaults

-funroll-loops or **-qunroll=auto**

Parameters

The following suboptions are for **-qunroll** only:

auto
> This suboption is equivalent to **-funroll-loops**.

yes
> This suboption is equivalent to **-funroll-all-loops**.

no Instructs the compiler to not unroll loops.

n   Instructs the compiler to unroll loops by a factor of *n*. In other words, the body of a loop is replicated to create *n* copies and the number of iterations is reduced by a factor of *1/n*. The **-qunroll=n** option specifies a global unroll factor that affects all loops that do not already have an unroll pragma. The value of *n* must be a positive integer.

Specifying **#pragma unroll(1)** or **-qunroll=1** disables loop unrolling, and is equivalent to specifying **#pragma nounroll** or **-qnounroll**. If *n* is not specified and if **-qhot**, **-qsmp**, **-O4**, or **-O5** is specified, the optimizer determines an appropriate unrolling factor for each nested loop.

The compiler might limit unrolling to a number smaller than the value you specify for *n*. This is because the option form affects all loops in source files to which it applies and large unrolling factors might significantly increase compile time without necessarily improving runtime performance. To specify an unrolling factor for particular loops, use the #pragma form in those loops.

Specifying **-qunroll** without any suboptions is equivalent to **-qunroll=yes**.

Usage

The pragma overrides the 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 unrolled.

Only one pragma can be specified on a loop.

The pragma affects only the loop that follows it. An inner nested loop requires a **#pragma unroll** directive to precede it if the wanted loop unrolling strategy is different from that of the prevailing option.

Predefined macros

None.

Related information:

"#pragma unroll, #pragma nounroll" on page 252

-fvisibility (-qvisibility)
Category

Optimization and tuning

Pragma equivalent

- **-fvisibility**: #pragma GCC visibility push (default | protected | hidden)
- **-qvisibility**: #pragma GCC visibility push (default | protected | hidden)

#pragma GCC visibility pop

Purpose

Specifies the visibility attribute for external linkage entities in object files. The external linkage entities have the visibility attribute that is specified by the **-fvisibility** option if they do not get visibility attributes from pragma directives, explicitly specified attributes, or propagation rules.

Syntax

```
►►── -f─visibility─=──┬─default──┬──────────────────────────────────────────►◄
                      ├─hidden───┤
                      └─protected┘
```

```
►►── -q─visibility──=──┬─default──┬────────────────────────────────◄◄
                       ├─hidden───┤
                       └─protected┘
```
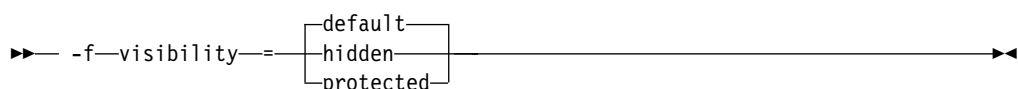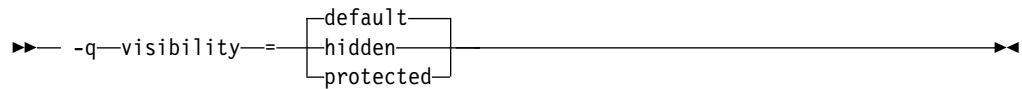
Defaults

-fvisibility=default or **-qvisibility=default**

Parameters

default
    Indicates that the affected external linkage entities have the default visibility attribute. These entities are exported in shared libraries, and they can be preempted.

protected
    Indicates that the affected external linkage entities have the protected visibility attribute. These entities are exported in shared libraries, but they cannot be preempted.

hidden
    Indicates that the affected external linkage entities have the hidden visibility attribute. These entities are not exported in shared libraries, but their addresses can be referenced indirectly through pointers.

    The **-qvisibility=internal** option is not supported; use the **-qvisibility=hidden** option instead.

Usage

The **-fvisibility** option globally sets visibility attributes for external linkage entities to describe whether and how an entity defined in one module can be referenced or used in other modules. Entity visibility attributes affect entities with external linkage only, and cannot increase the visibility of other entities. Entity preemption occurs when an entity definition is resolved at link time, but is replaced with another entity definition at run time.

Predefined macros

None.

Examples

To set external linkage entities with the protected visibility attribute in compilation unit myprogram.c, compile myprogram.c with the **-fvisibility=protected** option.

```
xlc myprogram.c -fvisibility=protected -c
```

All the external linkage entities in the myprogram.c file have the protected visibility attribute if they do not get visibility attributes from pragma directives, explicitly specified attributes, or propagation rules.

Related information
- "-shared (-qmkshrobj)" on page 220
- "Supported GCC pragmas" on page 240
- "Using visibility attributes (IBM extension)" in the *XL C/C++ Optimization and Programming Guide*

- "The visibility variable attribute (IBM extension)", "The visibility function attribute (IBM extension)", "The visibility type attribute (C++ only) (IBM extension)", and "The visibility namespace attribute (C++ only) (IBM extension)" in the *XL C/C++ Language Reference*

-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.

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

Syntax

Defaults

If **-g** is not specified, **-g0** takes effect, which means that the compiler does not generate any debug information or preserve program state.

If **-g** is specified, the default value is as follows:
- When no optimization is enabled (**-qnoopt**), **-g** is equivalent to **-g9**.
- When the **-O2** optimization level is in effect, **-g** is equivalent to **-g2**.

Parameters

-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. This option is equivalent to **-qlinedebug**.

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

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

-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.
- Debugging inlined functions is supported.

-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.
- Debugging inlined functions is supported.

Usage

When no optimization is enabled, the debugging information is always available if you specify **-g2**, **-g8**, or **-g9**.

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

When you specify **-g** with **-fstandalone-debug**, the compiler generates the debugging information for all symbols whether or not these symbols are referenced by the program. When you specify **-g** with **-fno-standalone-debug**, the compiler generates debugging information only for symbols that are referenced by the program.

Examples

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

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

The following command compiles `myprogram.c` with optimization level **-O2** and uses **-g8** to gain debugging information at optimization:

```
xlc myprogram.c -O2 -g8
```

Related information
- "-fstandalone-debug" on page 107
- "-qlinedebug" on page 169

- "-qfullpath" on page 151
- "-O, -qoptimize" on page 80
- "-qkeepparm" on page 167

-include (-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

```
►►──-include─file─────────────────────────────────────────►◄
```

```
          ┌─noinclude─────┐
►►──-q──┴── include──=─file─┴──────────────────────────────►◄
```

Defaults

None.

Parameters

file
    The header file to be included in the compilation units being compiled.

Usage

Firstly, *file* is searched in the preprocessor's working directory. If *file* is not found in the preprocessor's working directory, it is searched for in the search chain of the **#include** directive. If multiple **-include (-qinclude)** options are specified, the files are included in order of appearance on the command line.

Predefined macros

None.

Examples

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

```
xlc -include test1.h -include test2.h test.c
```

Related information
- "Directory search sequence for included files" on page 9

-isystem (-qc_stdinc) (C only)
Category

Compiler customization

Pragma equivalent

None.

Purpose

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

Syntax

```
►►──-isystem──dir──────────────────────────────────────────►◄
```

```
►►── -q──c_stdinc──=──┬───┬──▼──directory_path──┬───┬──────►◄
                      └─"─┘        ↑__:__        └─"─┘
```

Defaults

By default, the compiler searches the directory specified in the configuration file for the XL C header files (this is normally /opt/ibm/xlC/13.1.6/include/).

Parameters

dir
> The directory for the compiler to search for XL C header files. The search directories are after all directories specified by the **-I** option but before the standard system directories. The *dir* can be a relative or absolute path.

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 included files" on page 9 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 **-nostdinc** or **-nostdinc++** (**-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:

```
xlc myprogram.c -isystem mypath/headers1 -isystem mypath/headers2
```

Related information
- "-isystem (-qgcc_c_stdinc) (C only)" on page 125
- "-qstdinc, -qnostdinc (-nostdinc, -nostdinc++)" on page 206
- "-include (-qinclude)" on page 122
- "Directory search sequence for included files" on page 9
- "Specifying compiler options in a configuration file" on page 6
- "-I" on page 78

-isystem (-qcpp_stdinc) (C++ only)
Category

Compiler customization

Pragma equivalent

None.

Purpose

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

Syntax

►►──-isystem──*dir*──►◄

►►── -q──cpp_stdinc──=──┬──"──┬──▼──*directory_path*──┬──────┬──"──┬──►◄
                                      └──:──┘

Defaults

By default, the compiler searches the directory specified in the configuration file for the XL C++ header files (this is normally /opt/ibm/xlC/13.1.6/include/).

Parameters

dir
> The directory for the compiler to search for XL C++ header files. The search directories are after all directories specified by the **-I** option but before the standard system directories. The *dir* can be a relative or absolute path.

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 included files" on page 9 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 **-nostdinc** or **-nostdinc++** (**-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:

```
xlc myprogram.C -isystem mypath/headers1 -isystem mypath/headers2
```

Related information
- "-isystem (-qgcc_cpp_stdinc) (C++ only)" on page 127
- "-qstdinc, -qnostdinc (-nostdinc, -nostdinc++)" on page 206
- "-include (-qinclude)" on page 122
- "Directory search sequence for included files" on page 9
- "Specifying compiler options in a configuration file" on page 6
- "-I" on page 78

-isystem (-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

▶▶──-isystem──*dir*──◀◀

```
                                              ┌────: ──────┐
                                              ▼            │
►►── -q─gcc_c_stdinc──=──┬──┬──┬──directory_path──┬──┬──┬──────────────►◄
                         └─"─┘                    └─"─┘
```

Defaults

By default, the compiler searches the directory specified in the configuration file.

Parameters

dir
> The directory for the compiler to search for GNU C header files. The search directories are after all directories specified by the **-I** option but before the standard system directories. The *dir* can be a relative or absolute path.

directory_path
> The path for the directory where the compiler should search for the GNU 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 GNU C headers, you use a configuration file to do so; see "Directory search sequence for included files" on page 9 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 **-nostdinc** or **-nostdinc++** (**-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:

```
xlc myprogram.c -isystem mypath/headers1 -isystem mypath/headers2
```

Related information
- "-isystem (-qc_stdinc) (C only)" on page 123
- "-qstdinc, -qnostdinc (-nostdinc, -nostdinc++)" on page 206
- "-include (-qinclude)" on page 122
- "Directory search sequence for included files" on page 9
- "Specifying compiler options in a configuration file" on page 6
- "-I" on page 78

-isystem (-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

```
►►──-isystem─dir─────────────────────────────────────────────►◄
```

```
                                    ┌─ : ─────────┐
                                    ▼             │
►►── -q─gcc_cpp_stdinc──=──┬───┬──directory_path──┴──┬───┬──────►◄
                          └─"─┘                       └─"─┘
```

Defaults

By default, the compiler searches the directory specified in the configuration file.

Parameters

dir
> The directory for the compiler to search for GNU C++ header files. The search directories are after all directories specified by the -I option but before the standard system directories. The *dir* can be a relative or absolute path.

directory_path
> The path for the directory where the compiler should search for the GNU 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 GNU C++ headers, you use a configuration file to do so; see "Directory search sequence for included files" on page 9 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 **-nostdinc** or **-nostdinc++** (**-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:

```
xlc myprogram.C -isystem mypath/headers1 -isystem mypath/headers2
```

Related information
- "-isystem (-qcpp_stdinc) (C++ only)" on page 124
- "-qstdinc, -qnostdinc (-nostdinc, -nostdinc++)" on page 206
- "-include (-qinclude)" on page 122
- "Directory search sequence for included files" on page 9
- "Specifying compiler options in a configuration file" on page 6
- "-I" on page 78

-l

Category

Linking

Pragma equivalent

None.

Purpose

Searches for the specified library file. The linker searches for *libkey.so*, and then
libkey.a if *libkey.so* is not found.

Syntax

▶▶── -l─*key*──◀◀

Defaults

The compiler default is to search only some of the compiler runtime libraries. The
default configuration file specifies the default library names to search for with the
-l compiler option, and the default search path for libraries with the **-L** compiler
option.

The C and C++ runtime libraries are automatically added.

Parameters

key
    The name of the library minus the lib and *.a* or *.so* 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 `libmylibrary.so` or `libmylibrary.a` that is found in the `/usr/mylibdir` directory, enter the following command. Preference is given to `libmylibrary.so` over `libmylibrary.a`.

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

Related information
- "-L" on page 79
- "Specifying compiler options in a configuration file" on page 6

-maltivec (-qaltivec)

Category

Language element control

Pragma equivalent

None.

Purpose

Enables the compiler support for vector data types and operators.

Syntax

```
                          ┌─=le─┐
              ┌─altivec───┼─=be─┤
►►── -m───────┴─noaltivec─┴──────────────────────────────►◄


                          ┌─=le─┐
              ┌─altivec───┼─=be─┤
►►── -q───────┴─noaltivec─┴──────────────────────────────►◄
```

Defaults

`-maltivec=le(-qaltivec=le)`

Parameters

be   Specifies big endian element order. Vectors are laid out in vector registers from left to right, so that element 0 is the leftmost element in the register.

le   Specifies little endian element order. Vectors are laid out in vector registers from right to left, so that element 0 is the rightmost element in the register.

Usage

The **-maltivec (-qaltivec)** option takes effect only when you include the `altivec.h` file and set or imply **-mcpu** to be an architecture that supports vector instructions. Otherwise, the compiler ignores **-maltivec (-qaltivec)** and issues a warning message.

The **-maltivec (-qaltivec)** option affects the following categories of functions:
- Vector Multimedia Extension (VMX) load and store built-in functions
- Vector Scalar Extension (VSX) load and store built-in functions
- The nonload and nonstore built-in functions referring to the vector element order

The following list shows all the functions affected:
- Load functions
  - VMX load functions: **vec_ld**
  - VSX load functions: **vec_xld2**, **vec_xlw4**, and **vec_xl**
- Store functions
  - VMX store functions: **vec_st**
  - VSX store functions: **vec_xstd2**, **vec_xstw4**, and **vec_xst**
- Nonload and nonstore functions: **__vpermxor**, **vec_extract**, **vec_insert**, **vec_mergee**, **vec_mergeh**, **vec_mergel**, **vec_mergeo**, **vec_pack**, **vec_perm**, **vec_promote**, **vec_splat**, **vec_unpackh**, and **vec_unpackl**

Predefined macros

__ALTIVEC__ is defined to 1 and __VEC__ is defined to 10206 when **-maltivec (-qaltivec)** is in effect; otherwise, they are undefined.

__VEC_ELEMENT_REG_ORDER__ is defined to __ORDER_LITTLE_ENDIAN__ when **-maltivec=le (-qaltivec=le)** is in effect, or to __ORDER_BIG_ENDIAN__ when **-maltivec=be (-qaltivec=be)** is in effect.

Examples
- To enable compiler support for vector programming, enter the following command:

```
xlc myprogram.c -mcpu=pwr8 -maltivec
```

- To change the vector element sequence to big endian element order in registers, enter the following command:

```
xlc myprogram.c -maltivec=be
```

Related information
- "-mcpu (-qarch)" on page 131
- "Vector built-in functions" on page 365
- Vector types (IBM extension)
- "-qsimd" on page 198
- *AltiVec Technology Programming Interface Manual*, available at http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf

-mcpu (-qarch)
Category

Optimization and tuning

Pragma equivalent

None.

Purpose

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

Syntax

```
                ┌─=power8─┐
                ├─=pwr8──┤
►►── -m─cpu──┤─=pwr9──┤──────────────────────────────────────►◄
                └─=power9─┘


                ┌─=──pwr8─┐
►►── -q─arch──┤─=──pwr9─┤────────────────────────────────────►◄
                └─=──auto─┘
```
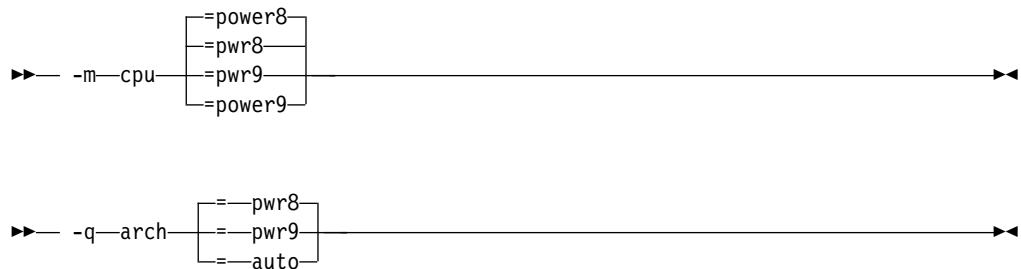
Defaults
* **-mcpu=pwr8**, **-mcpu=power8**, or **-qarch=pwr8**
* **-qarch=auto** when **-04** or **-05** is in effect

Parameters

auto
> Automatically detects the specific architecture of the compiling machine. It assumes that the execution environment will be the same as the compilation environment. This option is implied if the **-04** or **-05** option is set or implied. You can specify the **auto** suboption with **-qarch** only.

power8
> Produces object code containing instructions that run on the POWER8® or POWER9 hardware platforms. You can specify this suboption with **-mcpu** only.

power9
> Produces object code containing instructions that utilize the POWER9 technology. You can specify this suboption with **-mcpu** only.

pwr8
> Produces object code containing instructions that run on the POWER8 or POWER9 hardware platforms.

pwr9
> Produces object code containing instructions that utilize the POWER9 technology.

Usage

For any given **-mcpu** or **-qarch** setting, the compiler defaults to a specific, matching **-mtune** or **-qtune** setting, which can provide additional performance improvements. For detailed information about using **-mcpu (-qarch)** and **-mtune (-qtune)** together, see "-mtune (-qtune)."

Predefined macros

See "Macros related to architecture settings" on page 310 for a list of macros that are predefined by **-mcpu (-qarch)** suboptions.

Examples

To specify that the executable program `testing` compiled from `myprogram.c` is to run on a computer with VSX instruction support, for example, power8, enter:

```
xlc -o testing myprogram.c -mcpu=pwr8
```

Related information

- -qprefetch
- -qfloat
- "-mtune (-qtune)"
- "Macros related to architecture settings" on page 310
- "Optimizing your applications" in the *XL C/C++ Optimization and Programming Guide*

-mtune (-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. Allows specification of a target SMT mode to direct optimizations for best performance in that mode.

Syntax

```
►►── -m─tune─=─┬─power8─┬──────────────────────────────►◄
               ├─pwr8───┤
               ├─pwr9───┤
               └─power9─┘
```

```
                              ┌─balanced─┐
►►── -q─tune──=──┼─pwr8─────┤              ┌─st───────┐              ──►◄
                 ├─pwr9─────┤   └─:──┼─balanced─┤
                 └─auto─────┘        ├─smt2─────┤
                                     ├─smt4─────┤
                                     └─smt8─────┘
```
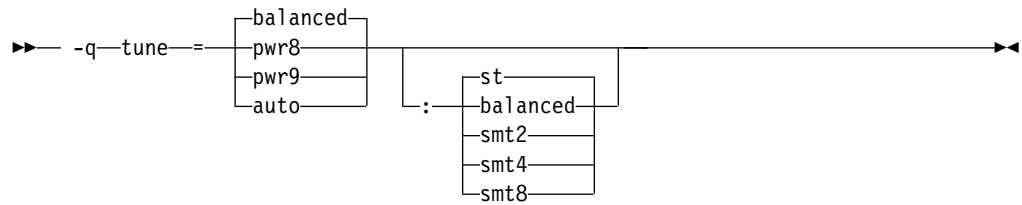
Defaults

-mtune=pwr8 or **-mtune=power8**

-qtune=balanced:balanced when no valid **-qarch** setting is in effect. Otherwise, the default depends on the effective **-qarch** setting. You can find details in Acceptable -qarch and -qtune combinations.

Parameters for CPU suboptions

The following CPU suboptions allow you to specify a particular architecture for the compiler to target for best performance:

auto
> Optimizations are tuned for the platform on which the application is compiled. You can specify the **auto** suboption with **-qtune** only.

balanced
> Optimizations are tuned across a selected range of recent hardware. You can specify the **balanced** suboption with **-qtune** only.

power8
> Optimizations are tuned for the POWER8 hardware platform. You can specify this suboption with **-mtune** only.

power9
> Optimizations are tuned to utilize the POWER9 technology. You can specify this suboption with **-mtune** only.

pwr8
> Optimizations are tuned for the POWER8 hardware platform.

pwr9
> Optimizations are tuned to utilize the POWER9 technology.

Parameters for SMT suboptions

The following simultaneous multithreading (SMT) suboptions allow you to optionally specify an execution mode for the compiler to target for best performance. You can specify these SMT suboptions with **-qtune** only.

balanced
> Optimizations are tuned for performance across various SMT modes for a selected range of recent hardware.

st Optimizations are tuned for single-threaded execution.

smt2
> Optimizations are tuned for SMT2 execution mode (two threads).

smt4
> Optimizations are tuned for SMT4 execution mode (four threads).

smt8

Optimizations are tuned for SMT8 execution mode (eight threads).

Usage

By arranging (scheduling) the generated machine instructions to take maximum advantage of hardware features such as cache size and pipelining, **-mtune** or **-qtune** can improve performance. It only has an effect when used in combination with options that enable optimization.

Although changing the **-mtune** or **-qtune** setting may affect the performance of the resulting executable, it has no effect on whether the executable can be executed correctly on a particular hardware platform.

You can find the acceptable combinations of **-qarch** and **-qtune** in the following table.

Table 21. Acceptable **-qarch** *and* **-qtune** *combinations*

-qarch option	Default -qtune setting	Available -qtune CPU settings	Available -qtune SMT settings
pwr8	pwr8:st	auto ∣ pwr8 ∣ pwr9 ∣ balanced	balanced ∣ st ∣ smt2 ∣ smt4 ∣ smt8
pwr9	pwr9:st	auto ∣ pwr9 ∣ balanced	balanced ∣ st ∣ smt2 ∣ smt4 ∣ smt8

Predefined macros

None.

Examples

The executable program `testing` is compiled from `myprogram.c`. To specify `testing` is to be optimized for a POWER8 hardware platform, enter:

```
xlc -o testing myprogram.c -mtune=pwr8
```

The executable program `testing` is compiled from `myprogram.c`. To specify `testing` is to be optimized for a POWER8 hardware platform and configured for the SMT4 mode, enter:

```
xlc -o testing myprogram.c -qtune=pwr8:smt4
```

Related information

- "-mcpu (-qarch)" on page 131
- "Optimizing your applications" in the *XL C/C++ Optimization and Programming Guide*

-o

Category

Output control

Pragma equivalent

None.

Purpose

Specifies a name for the output object, assembler, executable, or preprocessed file.

Syntax

►►── -o──*path*──◄◄

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. *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.
>
> You cannot specify a file name with a C or C++ source file suffix (.C, .c, or .cpp), such as `myprog.c`; this results in an error and neither the compiler nor the linker is invoked.

Usage

If you use the **-c** option with **-o**, you can compile only one source file at a time. In this case, if more than one source file name is specified, the compiler issues a warning message and ignores **-o**.

The **-P** and **-fsyntax-only (-qsyntaxonly)** options override the **-o** option.

Predefined macros

None.

Examples

To compile `myprogram.c` so that the resulting executable is called `myaccount`, enter:

```
xlc myprogram.c -o myaccount
```

To compile `test.c` to an object file only and name the object file `new.o`, enter:

```
xlc test.c -c -o new.o
```

Related information
- "-c" on page 91
- "-E" on page 75
- "-P" on page 84
- "-fsyntax-only (-qsyntaxonly)" on page 109

-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

```
►►─┬─ -p ──────────────────────┬────────────────────────────────►◄
   ├─ -pg ─────────────────────┤
   └─ -q ─ profile ─ = ─┬─ p ──┤
                        └─ pg ─┘
```

Defaults

Not applicable.

Usage

When you are compiling and linking in separate steps, you must specify the profiling option in both steps.

Predefined macros

None.

Examples

To compile myprogram.c to include profiling data, enter:

```
xlc myprogram.c -p
```

Remember to compile *and* link with one of the profiling options. For example:

```
xlc myprogram.c -p -c
xlc myprogram.o -p -o program
```

Related information
- See your operating system documentation for more information on the **gprof** command.
- For details about the GCC options **-p** and **-pg**, see the GCC online documentation at http://gcc.gnu.org/onlinedocs/.

-qaggrcopy
Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Enables destructive copy operations for structures and unions.

Syntax

```
►►── -q──aggrcopy──=──┬─nooverlap─┬──────────────────────────►◄
                      └─overlap───┘
```

Defaults

-qaggrcopy=nooverlap

Parameters

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

Predefined macros

None.

-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

```
►►── -q────asm_as──=──┬─path──────────────────┬────────────────►◄
                      └─"─path──┬──────┬──"────┘
                                └─flags─┘
```

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 the following command:

```
xlc 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 the following command:

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

Related information

- "-fasm (-qasm)" on page 94

-qcache

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

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.

The **-qcache** option must be specified with **-O4**, **-O5**, or **-qipa**.

Syntax

```
                          :                        :
                       ┌──◄──┐                  ┌──◄──┐
►►─── -q─cache─=─┬─────┴─┬─level─=─┬─1─┬─────┴──┬────────────────────┬─►◄
                 │       │         ├─2─┤        ├─assoc─=─number──────┤
                 │       │         └─3─┘        ├─auto───────────────┤
                 │       └─type─=─┬─c─┐         ├─cost─=─cycles──────┤
                 │                ├─d─┤         ├─line─=─bytes───────┤
                 │                └─i─┘         └─size─=─Kbytes──────┘
```
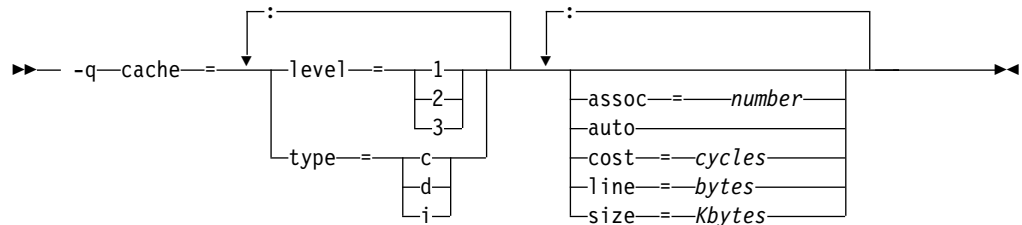
Defaults

Automatically determined by the setting of the **-mtune (-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
An integer representing the CPU 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   Table Lookaside Buffer (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 **-mtune (-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.

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 **-04** and **-05** optimization options automatically select the cache characteristics of the compiling machine. If you specify the **-qcache** option together with the **-04** or **-05** options, the option specified last takes precedence.
- Unless **-qcache=auto** is specified, you must specify both the **type** and **level** suboptions when you use the **-qcache** option. Otherwise, a warning message is issued.

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:

```
xlc -04 -qcache=type=c:level=1:size=8:line=64:assoc=2 file.c
```

Related information
- "-qcache" on page 138
- "-O, -qoptimize" on page 80

- "-mtune (-qtune)" on page 132
- "-qipa" on page 160
- "Optimizing your applications" in the *XL C/C++ Optimization and Programming Guide*

-qcheck

Category

Error checking and debugging

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 might result in slower application execution.

Syntax

```
         ┌─nocheck──────────────────────────┐
►►── -q ──┴─check──┬─────────────────────────┴──────────────────────►◄
                   │         ┌─:──────────┐
                   │      ┌──▼─ all ───┐
                   └─ = ──┼── bounds ───┤
                          ├── nobounds ─┤
                          ├── divzero ──┤
                          ├── nodivzero ┤
                          ├── nullptr ──┤
                          ├── nonullptr ┤
                          ├── stackclobber ┤
                          ├── nostackclobber ┤
                          ├── unset ────┤
                          └── nounset ──┘
```
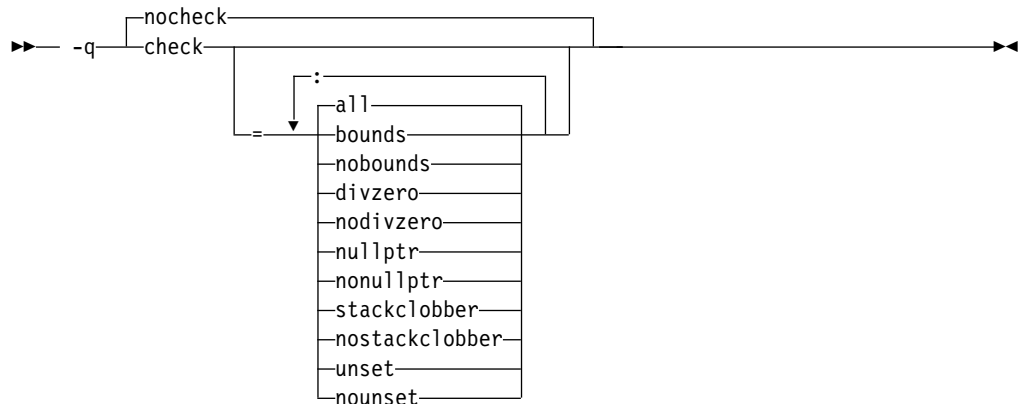
Defaults

-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.

`stackclobber | nostackclobber`

> Detects stack corruption of nonvolatile registers in the save area in user programs. This type of corruption happens only if any of the nonvolatile registers in the save area of the stack is modified.

`unset | nounset`

> Checks for automatic variables that are used before they are set. A trap will occur at run time if an automatic variable is not set before it is used.
>
> The **`-qinitauto`** option initializes automatic variables. As a result, the **`-qinitauto`** option hides uninitialized variables from the **`-qcheck=unset`** option.

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

Usage

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

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

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

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

Predefined macros

None.

Examples

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

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

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

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

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

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

```
void func4(char *p, int off, int value) {
  *(p+off)=value;
}
```

```
int foo() {
  int i;
  char boo[9];
  i=24;
  func4(boo, i, 66);
  /* Traps here */
  return 0;
}

int main() {
  foo();
}
```

Note: The offset is subject to change at different optimization level. When -O2 or lower optimization level is in effect, `func4` will clobber the save area of `foo` because `*(p+off)` is in the save area.

In function `factorial`, `result` is not initialized when n<=1. To detect an uninitialized variable in `factorial.c`, enter the following command:

```
xlc -g -O -qcheck=unset factorial.c
```

`factorial.c` contains the following code:

```
int factorial(int n) {
  int result;

  if (n > 1) {
    result = n * factorial(n - 1);
  }

  return result; /* line 8 */
}

int main() {
  int x = factorial(1);
  return x;
}
```

The compiler issues the following informational message during compile time and a trap occurs at line 8 during run time:

```
1500-099: (I) "factorial.c", line 8: "result" might be used before it is set.
```

Note: If you set **–qcheck=unset** at **noopt**, the compiler does not issue informational messages at compile time.

> **Related information**
>
> "-fstack-protector (-qstackprotect)" on page 104

-qcompact

Category

Optimization and tuning

Purpose

Avoids optimizations that increase code size.

Syntax

```
                  ┌─nocompact─┐
►►── -q───┴─compact──┴──────────────────────────────────────────►◄
```
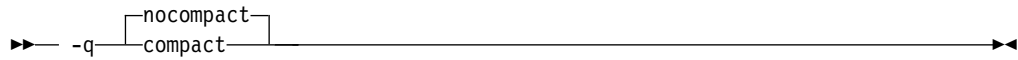
Defaults

-qnocompact

Usage

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

This option takes effect only when it is specified at the **-O2** optimization level, or higher.

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 the following command:

```
xlc myprogram.c -O -qcompact
```

-qcrt, -nostartfiles (-qnocrt)
Category

Linking

Pragma equivalent

None.

Purpose

When **-qcrt** is in effect, the system startup routines are automatically linked. When **-nostartfiles (-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 are linked.

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

Syntax

```
►►── -nostartfiles───────────────────────────────────────────►◄
```

```
              ┌─crt──┐
►►── -q───┴─nocrt─┴──────────────────────────────────────────►◄
```

Defaults

-qcrt
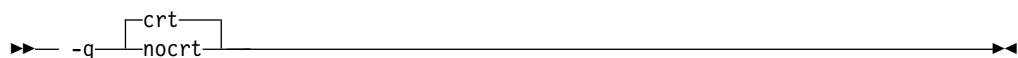
Predefined macros

None.

Related information
- "-qlib, -nodefaultlibs (-qnolib)" on page 167

-qdataimported, -qdatalocal, -qtocdata
Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Marks data as local or imported.

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 to be local. Alternatively, you can use the **-qtocdata** option to instruct the compiler to assume all variables to be 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 to be imported. Alternatively, you can use the **-qnotocdata** option to instruct the compiler to assume all variables to be imported.

Syntax

```
                 ┌─notocdata──────┐
                 │ ┌─dataimported─┐│
►►── -q ─┬───────┴─┴──────────────┴────────────────────────────┬──►◄
         │                    ┌─:──────────────┐                │
         │                    ▼                │                │
         │            └─=──────variable_name───┘                │
         │                                                      │
         ├─tocdata───────────────────────────────┐             │
         └─datalocal─────────────────────────────┘             │
                              ┌─:──────────────┐
                              ▼                │
                      └─=──────variable_name───┘
```
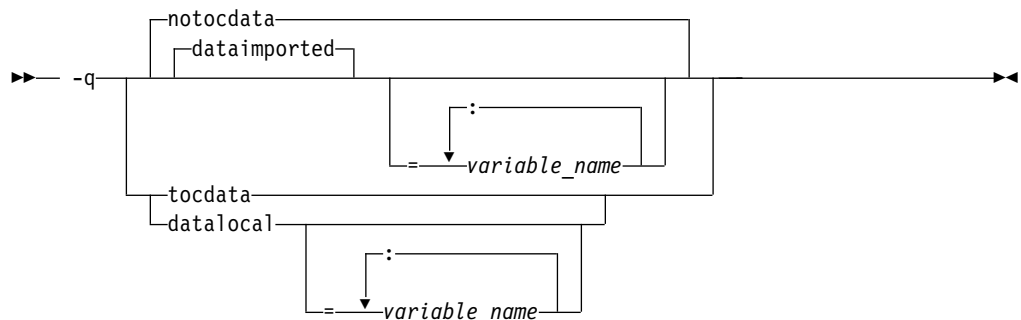
Defaults

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

Parameters

variable_name

> The name of a variable that the compiler should assume to be 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.
>
> 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.

-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

```
          ┌─nodirectstorage─┐
►►── -q──┴─directstorage───┴─────────────────────────────────────────────►◄
```

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.

-qfloat

Category

Floating-point and integer control

Purpose

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

Syntax
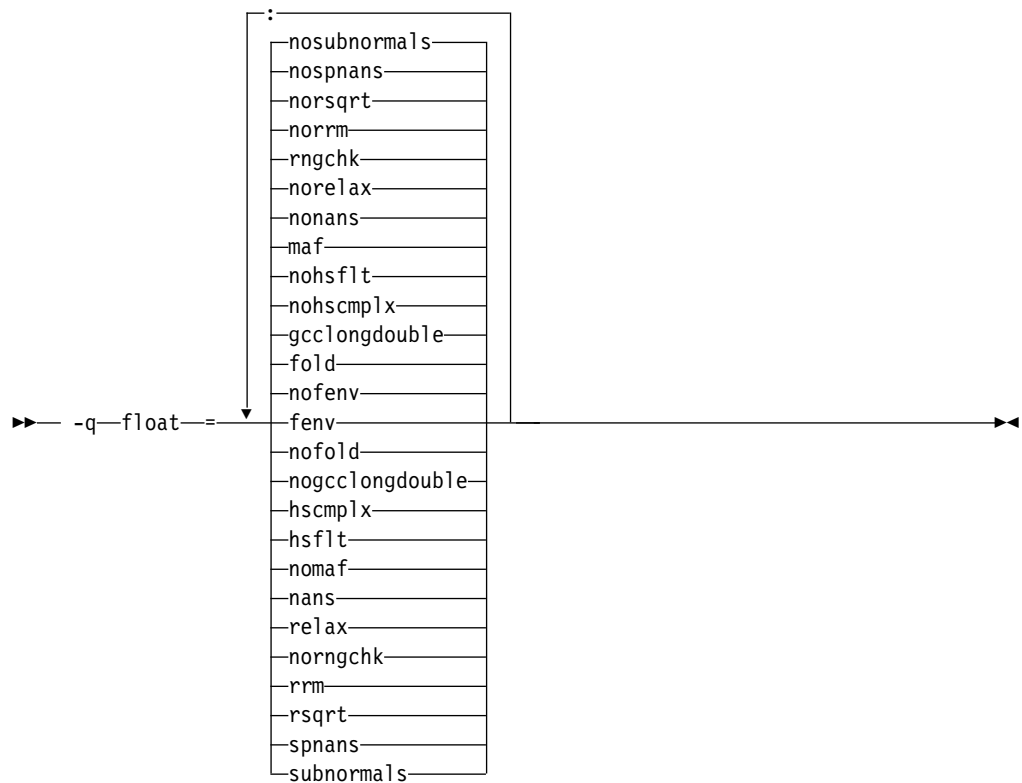
Defaults
- **-qfloat=nofenv:fold:gcclongdouble:nohscmplx:nohsflt:maf:nonans:norelax: rngchk:norrm:norsqrt:nospnans:nosubnormals**
- **-qfloat=rsqrt:norngchk** when **-qnostrict**, **-qstrict=nooperationprecision:noexceptions**, or the **-03** or higher optimization level is in effect.

Parameters

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.

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.

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 327 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. **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. Rounding towards negative infinity or positive infinity will be reversed for these operations. 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 | <u>nonans</u>
> 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 | <u>norelax</u>
> 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.

<u>rngchk</u> | 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 | <u>norrm</u>
> 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 | <u>norsqrt</u>
> 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 the **-03** or higher optimization level, **rsqrt** is enabled automatically. To disable it, also specify **-qstrict**, **-qstrict=nans**, **-qstrict=infinities**, **-qstrict=zerosigns**, or **-qstrict=exceptions**.

spnans | <u>**nospnans**</u>
> Generates extra instructions to detect signalling NaN on conversion from single-precision to double-precision.

subnormals | <u>**nosubnormals**</u>
> Specifies whether the code uses subnormal floating point values, also known as denormalized floating point values. Whether or not you specify this suboption, the behavior of your program will not change, but the compiler uses this information to gain possible performance improvements.

Note: For details about the relationship between **-qfloat** suboptions and their **-qstrict** counterparts, see "-qstrict" on page 208.

Usage

Using **-qfloat** suboptions other than the default settings might produce incorrect results in floating-point computations if the system does not meet all required conditions for a given suboption. Therefore, use this option only if the floating-point calculations involving IEEE floating-point values are manipulated and can properly assess the possibility of introducing errors in the program.

If the **-qstrict** | **-qnostrict** and **float** suboptions conflict, the last setting specified is used.

Predefined macros

None.

Examples

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

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

Related information
- "-mcpu (-qarch)" on page 131
- "-ftrapping-math (-qflttrap)" on page 111
- "-qstrict" on page 208
- "Handling floating-point operations" in the *XL C/C++ Optimization and Programming Guide*

-qfulldebug
Category

Error checking and debugging

Pragma equivalent

None.

Purpose

Generates the debugging information for all class members.

Syntax

```
            ┌─nofulldebug─┐
►►── -q─────┴─fulldebug───┴────────────────────────────────────────►◄
```

Defaults

-qnofulldebug

Usage

When the **-qfulldebug** option is in effect, the compiler produces the debugging information for all class members.

When the **-qnofulldebug** option is in effect, if a non-outlined member function has not been called, its debugging information is not generated.

Predefined macros

None.

-qfullpath

Category

Error checking and debugging

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

```
            ┌─nofullpath─┐
►►── -q─────┴─fullpath───┴─────────────────────────────────────────►◄
```

Defaults

-qnofullpath

Usage

If your executable file was moved to another directory, the debugger would be unable to find the file unless you provide a search path in the debugger. You can use **fullpath** to ensure that the debugger locates the file successfully.

Predefined macros

None.

Related information
- "-qlinedebug" on page 169
- "-g" on page 120

-qfuncsect
Category

Object code control

Purpose

Places instructions for each function in a separate section. Placing each function in its own section might reduce the size of your program because the linker can collect garbage per function rather than per object file.

When **-qnofuncsect** is in effect, each object file consists of a single text section combining all functions defined in the corresponding source file. You can use **-qfuncsect** to place each function in a separate section.

Syntax

```
                    ┌─nofuncsect─┐
►►── -q──┴─funcsect──┘──────────────────────────────►◄
```

Defaults

-qnofuncsect

Usage

Using multiple sections increases the size of the object file, but it can reduce the size of the final executable by allowing the linker to remove functions that are not called or that have been inlined by the optimizer at all places they are called.

The pragma directive must be specified before the first statement in the compilation unit.

Predefined macros

None.

-qhot

Category

Optimization and tuning

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.

Syntax
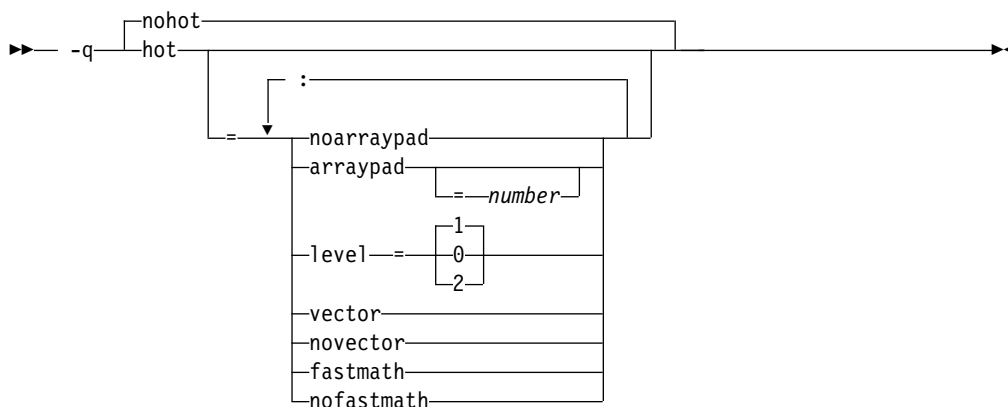
Defaults

- **-qnohot**
- **-qhot=noarraypad:level=0:novector:fastmath** when **-03** is in effect
- **-qhot=noarraypad:level=1:vector:fastmath** when **-qsmp**, **-04** or **-05** is in effect.
- Specifying **-qhot** without suboptions is equivalent to **-qhot=noarraypad:level=1:vector:fastmath**.

Parameters

arraypad | noarraypad
    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
> 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. To achieve more efficient cache utilization, it is recommended that pad values be multiples of the largest array element size, typically 4, 8, or 16.

level=0
> Performs a subset of the high-order transformations and sets the default to **novector:noarraypad:fastmath**.

level=1
> Performs the default set of high-order transformations.

level=2
> Performs the default set of high-order transformations and some more aggressive loop transformations. This option performs aggressive loop analysis and transformations to improve cache reuse and exploit loop parallelization opportunities.

vector | 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-precision 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.
>
> Because vectorization can affect the precision of your program results, if you are using **-O3** 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.

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 **-qreport** or **-qlistfmt** option is also specified. This `LOOP TRANSFORMATION SECTION` of the listing file also contains information about data prefetch insertion locations. In addition, when you use **-qprefetch=assistthread** to generate prefetching assist threads, a message `Assist thread for data prefetching was generated` also appears in the `LOOP TRANSFORMATION SECTION` of the listing file. Specifying **-qprefetch=assistthread** guides the compiler to generate aggressive data prefetching at optimization level **-03** or higher. For more information, see "-qreport" on page 188.

Predefined macros

None.

Related information
- "-mcpu (-qarch)" on page 131
- "-qsimd" on page 198
- "-qprefetch" on page 184
- "-qreport" on page 188
- "-qlistfmt" on page 171
- "-O, -qoptimize" on page 80
- "-qstrict" on page 208
- *Using the Mathematical Acceleration Subsystem (MASS)* in the *XL C/C++ Optimization and Programming Guide*
- "#pragma nosimd" on page 244

-qidirfirst

Category

Input control

Pragma equivalent

None.

Purpose

Searches for user included files in directories that are specified by the **-I** option before searching any other directories.

Syntax

```
          ┌─noidirfirst─┐
►►─── -q──┴─idirfirst───┴─────────────────────────────────────►◄
```

Defaults

-qnoidirfirst

Usage

This option only affects files that are included by the #include "*file_name*" directive or the **-include** option. This option has no effect on the search order for XL C/C++ or system header files. This option also has no effect on files that are included by absolute paths.

-qidirfirst is independent of the **-qnostdinc** option.

Predefined macros

None.

Examples

To compile myprogram.c and instruct the compiler to search for included files in /usr/tmp/myinclude first and then the directory in which the source file is located, use the following command:

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

Related information
- "-I" on page 78
- "-include (-qinclude)" on page 122
- "-qstdinc, -qnostdinc (-nostdinc, -nostdinc++)" on page 206
- "-isystem (-qc_stdinc) (C only)" on page 123
- "-isystem (-qcpp_stdinc) (C++ only)" on page 124
- "Directory search sequence for included files" on page 9

-qignerrno
Category

Optimization and tuning

Purpose

Allows the compiler to perform optimizations as if system calls would not modify errno.

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 option allows the compiler to perform optimizations without regard to what happens to errno.

Syntax

```
►►── -q──┬─noignerrno─┬────────────────────────────────────────────────────►◄
         └─ignerrno───┘
```

Defaults
- -qnoignerrno
- **-qignerrno** when the **-O3** or higher optimization level is in effect.

Usage

If you require both **-03** 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 **-qignerrno** is in effect; otherwise, it is undefined.

Related information
• "-O, -qoptimize" on page 80

-qinitauto

Category

Error checking and debugging

Purpose

Initializes uninitialized automatic variables to a specific value, for debugging purposes.

Syntax

```
                 ┌─noinitauto─────────────────┐
►►── -q──┴─initauto──=─hex_value─┘──────────────────────────────────────────►◄
```

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 that are not explicitly initialized when declared are cleared before they are 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.

Restrictions:
- Objects that are equivalenced, structure components, and array elements are not initialized individually. Instead, the entire storage sequence is initialized collectively.
- The **-qinitauto=hex_value** option does not initialize variable length arrays or memory allocated through the __alloca function.

Predefined macros

- __INITAUTO__ is defined to the least significant byte of the *hex_value* that is specified on the **-qinitauto** option or pragma; otherwise, it is undefined.
- __INITAUTO_W__ is defined to the byte *hex_value*, repeated four times, or to the word *hex_value*, which is specified on the **-qinitauto** option or pragma; otherwise, it is undefined.

For example:
- For option -qinitauto=0xABCD, the value of __INITAUTO__ is 0xCDu, and the value of __INITAUTO_W__ is 0x0000ABCDu.
- For option -qinitauto=0xCD, the value of __INITAUTO__ is 0xCDu, and the value of __INITAUTO_W__ is 0xCDCDCDCDu.

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%llX\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

Purpose

When used with **-O2** or higher optimization, inlines glue code that optimizes external function calls in your application.

Glue code or Procedure Linkage Table 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.

Syntax

```
►►── -q──┬─inlglue───┬──────────────────────────────────────►◄
         └─noinlglue─┘
```

Defaults

- **-qinlglue**

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 143
- "-mtune (-qtune)" on page 132

-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

```
                  ┌─noipa─────────────────────────┐
►►── -q──────┬────┴─ipa──────────────────────┬────────────►◄
                                └──=──┬─object───┬──┘
                                      └─noobject─┘
```

-qipa link-time syntax

```
                  ┌─noipa──────────────────────────────────────┐
►►── -q───────┬───┴─ipa────────────────────────────────────┬──────►◄
              │
              │        ┌──────────────:──────────────┐
              │        │              ┌─,─┐          │
              └──=──▼──┬─exits──=──▼──function_name───┤
                       │                        ┌─,─┐
                       ├─infrequentlabel──=──▼──label_name──┤
                       │              ┌─1─┐
                       ├─level──=──┬──┼─0─┤
                       │           └──┴─2─┘
                       ├─list──┬────────────────────┤
                       │       └──=──┬─file_name─┐
                       │             ├─long───────┤
                       │             └─short──────┘
                       │              ┌─,─┐
                       ├─lowfreq──=──▼──function_name──┤
                       │           ┌─unknown──┐
                       ├─missing──=──┼─safe─────┤
                       │           ├─isolated─┤
                       │           └─pure─────┘
                       │             ┌─medium─┐
                       ├─partition──=──┼─small──┤
                       │             └─large──┘
                       │                       ┌─,─┐
                       ├─┬─isolated─┬──=──▼──function_name──┤
                       │ ├─pure─────┤
                       │ ├─safe─────┤
                       │ └─unknown──┘
                       └─file_name──────────────────────────┘
```
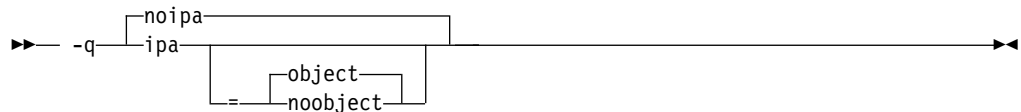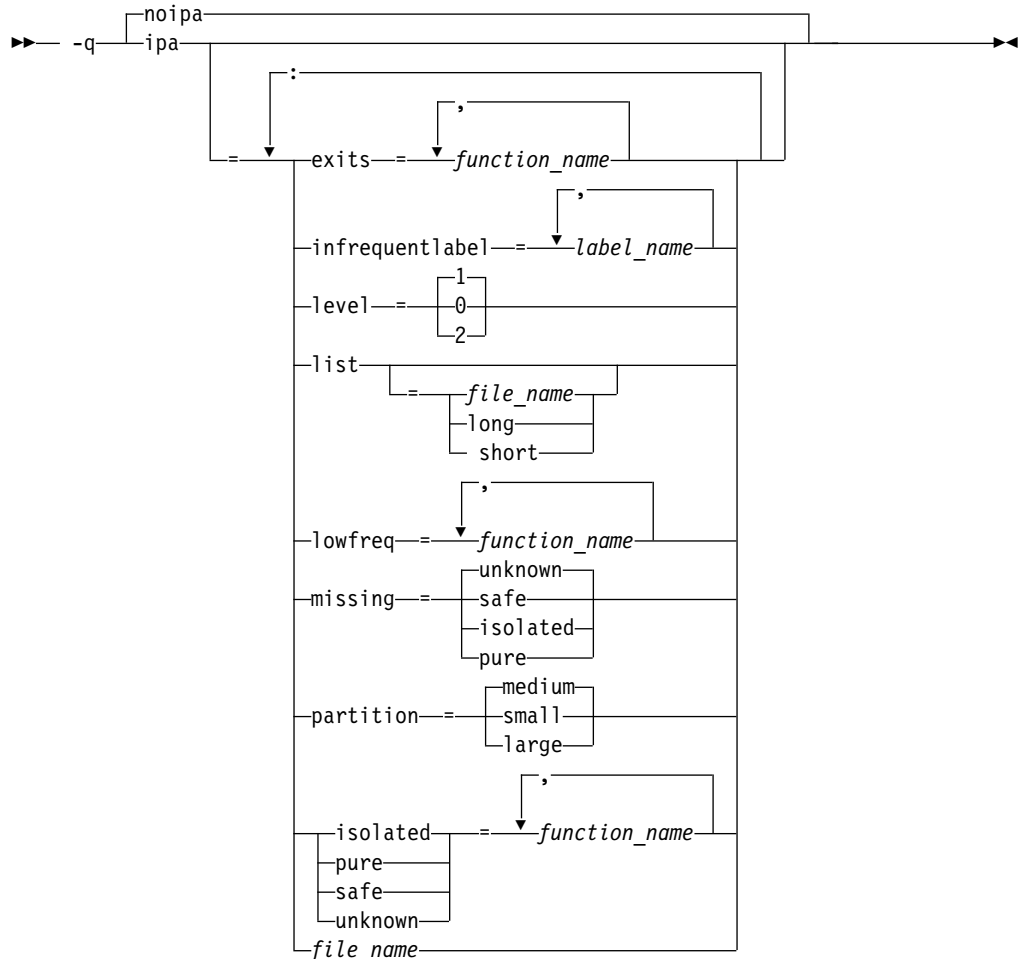
Defaults

- **-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 step in the same compiler invocation, or during a separate link step only:

clonearch | noclonearch
    This suboption is no longer supported. Consider using **-qtune=balanced**.

cloneproc | nocloneproc
    This suboption is no longer supported. Consider using **-qtune=balanced**.

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:

<u>short</u> 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.

<u>unknown</u>

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**
- <u>medium</u>
- **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**.

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 **-02**. For additional performance benefits, you can also specify the **-finline-functions (-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:

```
xlc -c *.c -qipa
xlc -o product *.o -qipa
```

Here is how you might compile the same set of files, improving the optimization of the second compilation, and the speed of the first compile step. Assume that there exist a set of routines, user_trace1, user_trace2, and user_trace3, which are rarely executed, and the routine user_abort that exits the program:

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

Related information
- "-finline-functions (-qinline)" on page 99
- "-qisolated_call"
- "#pragma execution_frequency" on page 242
- "-S" on page 86
- "Optimizing your applications" in the *XL C/C++ Optimization and Programming Guide*
- Runtime environment variables

-qisolated_call
Category

Optimization and tuning

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

```
►►── -q──isolated_call──=──┬──function──┬──────────────────────────────────►◄
                           │    ◄─:◄     │
                           └─────────────┘
```
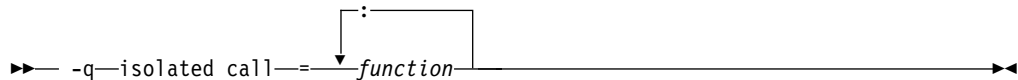
Defaults

Not applicable.

Parameters

function
    The name of a function that does not have side effects or does not rely on functions or processes that have side effects. *function* is a primary expression that can be an identifier, operator function, conversion function, or qualified name. An identifier must be of type function or a `typedef` of function. ▶ **C++** If the name refers to an overloaded function, all variants of that function are marked as isolated calls. **C++** ◄

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 nonvolatile external objects and return a result that depends on the nonvolatile 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.

Predefined macros

None.

Examples

To compile `myprogram.c`, specifying that the functions `myfunction(int)` and `classfunction(double)` do not have side effects, enter:

```
xlc myprogram.c -qisolated_call=myfunction:classfunction
```

Related information
* "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

```
                ┌─nokeepparm─┐
►►── -q─────────┴─keepparm───┴──────────────────────────────────►◄
```

Defaults

-qnokeepparm

Usage

Specifying **-qkeepparm** that the values of incoming parameters are available to tools, such as debuggers, by preserving those values on the stack. However, this may negatively affect application performance.

Predefined macros

None.

Related information
- "-O, -qoptimize" on page 80

-qlib, -nodefaultlibs (-qnolib)

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 **-nodefaultlibs (-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

►►──-nodefaultlibs───►◄

►►── -q──┬─lib──┬───►◄
          └─nolib─┘

Defaults

-qlib

Usage

Using **-nodefaultlibs (-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 **-nostartfiles (-qnocrt)** is also specified.

Note: 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 **-nodefaultlibs (-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:

```
xlc myprogram.c -nodefaultlibs -lxlopt
```

Related information
• "-qcrt, -nostartfiles (-qnocrt)" on page 144

-qlibansi

Category

Optimization and tuning

Pragma equivalent

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

```
             ┌─nolibansi─┐
►►─── -q─────┴─libansi───┴──────────────────────────────────────►◄
```

Defaults

-qnolibansi

Predefined macros

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

-qlinedebug

Category

Error checking and debugging

Pragma equivalent

None.

Purpose

Generates only line number and source file name information for a debugger.

When **-qlinedebug** is in effect, the compiler produces minimal debugging information, so the resulting object size is smaller than that produced by the **-g** debugging option. You can use the debugger to step through the source code, but you will not be able to see or query variable information. The traceback table, if generated, will include line numbers.

-qlinedebug is equivalent to **-g1**.

Syntax

```
►►── -q──┬─nolinedebug─┬──────────────────────────────────◄◄
         └─linedebug───┘
```

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:

```
xlc myprogram.c -o testing -qlinedebug
```

Related information
- "-g" on page 120
- "-O, -qoptimize" on page 80

-qlist

Category

Listings, messages, and compiler information

Purpose

Produces a compiler listing file that includes object and constant area sections.

Syntax

```
►►── -q──┬─nolist────────────────┬──────────────────────◄◄
         └─list──┬─────────────────────┬┘
                 └─=──┬─nooffset─┬──────┘
                      └─offset───┘
```

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

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 13.

You can use the object or assembly listing to help understand the performance characteristics of the generated code and to diagnose execution problems.

Predefined macros

None.

Examples

To compile `myprogram.c` and to produce a listing (.lst) file that includes object , enter:

```
xlc myprogram.c -qlist
```

-qlistfmt

Category

Listings, messages, and compiler information

Pragma equivalent

None.

Purpose

Creates a report in XML or HTML format to help you find optimization opportunities.

Syntax
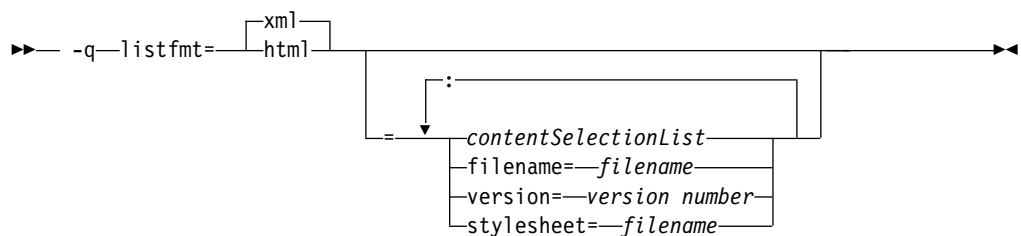
Defaults

This option is off by default. If none of the *contentSelectionList* suboptions is specified, all available report information is produced. For example, specifying **-qlistfmt=xml** is equivalent to **-qlistfmt=xml=all**.

Parameters

The following list describes **-qlistfmt** parameters:

<u>xml</u> | html
Instructs the compiler to generate the report 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" on page 299.

contentSelectionList
The following suboptions provide a filter to limit the type and quantity of information in the report:

data | <u>nodata</u>
Produces data reorganization information.

inlines | <u>noinlines</u>
Produces inlining information.

pdf | <u>nopdf</u>
Produces profile-directed feedback information.

transforms | <u>notransforms</u>
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 the foo.c file is compiled, 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,

```
xlc -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 supplied with XL C/C++ is `xlstyle.xsl`. This stylesheet renders the XML report to an easily read format when the report is viewed through 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 catalog (`XMLMessages-`*locale*`.xml` where *locale* refers to the locale set on the compilation machine) in the path specified by the **stylesheet** suboption. The stylesheet and message catalog are installed in the `/opt/ibm/xlC/13.1.6/listings/` directory.

For example, if `a.xml` is generated with **stylesheet=xlstyle.xsl**, both `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.

version

Specifies the major version of the content that will be generated. If you have written a tool that requires a certain version of this report, you must specify the version.

For example, IBM XL C/C++ for Linux, V13.1.6 creates reports at XML `v1.1`. 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 you specify both **-qlistfmt** and an option that enables inlining such as **-finline-functions(-qinline)**, the report shows which functions were inlined and why others were not inlined.
- When you specify both **-qlistfmt** and an option that enables loop unrolling, the report contains a summary of how program loops are optimized. The report also includes diagnostic information about why specific loops cannot be vectorized. To make **-qlistfmt** generate information about loop transformations, you must also specify at least one of the following options:
  - **-qhot**
  - **-qsmp**
  - **-O3** or higher
- When you specify both **-qlistfmt** and 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 you specify both **-qlistfmt** and **-qpdf**, which enables profiling, the report contains information about call and block counts and cache misses.
- When you specify both **-qlistfmt** and an option that produces data reorganizations such as **-qipa=level=2**, the report contains information about those reorganizations.

Predefined macros

None.

Examples

If you want to compile myprogram.c to produce an XML report that shows how loops are optimized, enter:

```
xlc -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:

```
xlc -finline-functions -qlistfmt=xml=inlines myprogram.c
```

Related information
- "-qreport" on page 188
- "genhtml" on page 299
- "Using compiler reports to diagnose optimization opportunities" in the *XL C/C++ Optimization and Programming Guide*

-qmaxmem
Category

Optimization and tuning

Purpose

Limits the amount of memory that the compiler allocates while performing specific, memory-intensive optimizations to the specified number of kilobytes.

Syntax

►►── -q─maxmem──=─*size_limit*──────────────────────────────►◄

Defaults
- **-qmaxmem=8192** when **-O2** is in effect.
- **-qmaxmem=-1** when the **-O3** or higher optimization level 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:

```
xlc myprogram.c -qmaxmem=16384
```

-qmakedep, -MD (-qmakedep=gcc)

Category

Output control

Pragma equivalent

None.

Purpose

Produces the dependency files that are used by the **make** tool for each source file.

The dependency output file is named with a .d suffix.

Syntax

```
►►──-q──makedep──────────────────────────────────────►◄
                 └──=──gcc──┘
```

Defaults

Not applicable.

Parameters

gcc
> The format of the generated **make** rule to match the GCC format: the dependency output file includes a single target that lists all of the main source file's dependencies.
>
> This suboption is equivalent to **-MD**.

If you specify -**qmakedep** with no suboption, the dependency output 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 that is named on the command line, a dependency output file is generated with the same name as the object file but with a .d suffix. Dependency output files are not created for any other types of

input files. If you use the **-o** option to rename the object file, the name of the
dependency output file is based on the name specified in the **-o** option. For more
information, see the Examples section.

The dependency output files generated by these options are not **make** description
files; they must be linked before they can be used with the **make** command. For
more information about 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
```

Include files are listed according to the search order rules for the `#include`
preprocessor directive, described in "Directory search sequence for included files"
on page 9. If the include file is not found, it is not added to the .d file.

Files with no include statements produce dependency output files that contain one
line listing only the input file name.

Predefined macros

None.

Examples

Example 1: To compile `mysource.c` and create a dependency output file named
`mysource.d`, enter:

```
xlc -c -qmakedep mysource.c
```

Example 2: To compile `foo_src.c` and create a dependency output file named
`mysource.d`, enter:

```
xlc -c -qmakedep foo_src.c -MF mysource.d
```

Example 3: To compile `foo_src.c` and create a dependency output file named
`mysource.d` in the `deps/` directory, enter:

```
xlc -c -qmakedep foo_src.c -MF deps/mysource.d
```

Example 4: To compile `foo_src.c` and create an object file named `foo_obj.o` and a
dependency output file named `foo_obj.d`, enter:

```
xlc -c -qmakedep foo_src.c -o foo_obj.o
```

Example 5: To compile `foo_src.c` and create an object file named `foo_obj.o` and a
dependency output file named `mysource.d`, enter:

```
xlc -c -qmakedep foo_src.c -o foo_obj.o -MF mysource.d
```

Example 6: To compile `foo_src1.c` and `foo_src2.c` to create two dependency
output files, named `foo_src1.d` and `foo_src2.d` respectively, enter:

```
xlc -c -qmakedep foo_src1.c foo_src2.c
```

Related information
- "-o" on page 134
- "Directory search sequence for included files" on page 9

- The **-M**, **-MD**, **-MF**, **-MG**, **-MM**, **-MMD**, **-MP**, **-MQ**, and **-MT** options that GCC provides. For details, see the GCC online documentation at http://gcc.gnu.org/onlinedocs/.

-qoffload

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Enables support for offloading OpenMP target regions to an NVIDIA GPU.

You can use the **#pragma omp target** OpenMP directive to define a target region.

For **-qoffload** to take effect, you must specify the **-qsmp** option to enable the support for OpenMP target regions.

Syntax

```
>>-- -q --+-nooffload-+------------------------------------------><
          '-offload---'
```

Defaults

-qnooffload

Usage

To use the **-qoffload** option, you must install the CUDA Toolkit. To install the CUDA Toolkit, use the Package Manager installation. The Runfile installation is currently not supported on Power processors. For instructions about Package Manager installation, see the NVIDIA CUDA Installation Guide for Linux (http://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html).

You must specify the **-qoffload** option on both the compile and link steps. If you do not specify **-qoffload**, target regions run on the host, and the compiler does not generate any device code.

The compiler optimizes device code by default. You can use the **-qoffload** option with **-O2**, **-O3**, **-Ofast**, and **-O3 -qhot**. You cannot use the **-qoffload** option with **-O4**, **-O5**, **-qipa**, **-qpdf1**, or **-qpdf2**.

Predefined macros

None.

Examples

To compile `myopenmpprogram.c`, enter the following command:

```
xlC -qsmp -qoffload myopenmpprogram.c -o myopenmpprogram
```

To compile `myopenmpprogram.c` into an object file and link it afterwards, enter the following commands:

```
xlC -c -qsmp -qoffload myopenmpprogram.c -o myopenmpprogram.o
xlC -qsmp -qoffload myopenmpprogram.o -o myopenmpprogram
```

Related information
- -qsmp
- -qtgtarch
- #pragma omp target
- Offloading computations to the NVIDIA GPUs in the *XL C/C++ Optimization and Programming Guide*

-qpath

Category

Compiler customization

Pragma equivalent

None.

Purpose

Specifies substitute path names for XL C/C++ components 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++ components and have the option of specifying which one you want to use. This option is preferred over the **-B** and **-t** options.

Syntax
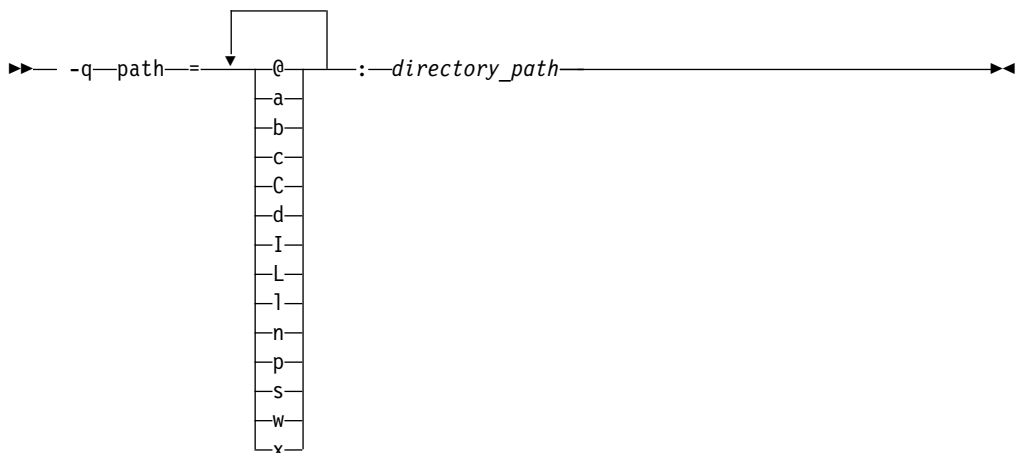
Defaults

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 names:

Parameter	Description	Component name
`GPU` @	The PTX assembler	ptxas
a	The assembler	as
b	The low-level optimizer	xlCcode
c, C	The C and C++ compiler front end	xlCentry
d	The disassembler	dis
I (uppercase i)	The high-level optimizer, compile step	ipa
L	The high-level optimizer, link step	ipa
l (lowercase L)	The linker	ld
`GPU` n	The NVIDIA C compiler, which is used as a device linker	nvcc
p	The preprocessor	xlCentry
`GPU` s	The XL intermediate language (W-Code) splitter	partitioner
`GPU` w	The XL intermediate language (W-Code) to NVVM-IR translator	wc2llvm
`GPU` x	The NVVM-IR to PTX translator	llvm2ptx

Usage

The **-qpath** option overrides the **-F**, **-t**, and **-B** options.

`GPU` To use **-qpath=@**, **-qpath=n**, **-qpath=s**, **-qpath=w**, or **-qpath=x**, you must specify the **-qoffload** option. `GPU`

Predefined macros

None.

Examples

To compile myprogram.c using a substitute compiler front end in /lib/tmp/mine/, enter the command:

```
xlc myprogram.c -qpath=c:/lib/tmp/mine/
```

To compile `myprogram.c` using a substitute linker in `/lib/tmp/mine/`, enter the command:

```
xlc myprogram.c -qpath=l:/lib/tmp/mine/
```

Related information

- "-B" on page 72
- "-F" on page 77
- "-t" on page 228

-qpdf1, -qpdf2
Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Tunes optimizations through *profile-directed feedback* (PDF), where results from sample program execution are used to improve optimization near conditional branches and in frequently executed code sections.

Optimizes an application for a typical usage scenario based on an analysis of how often branches are taken and blocks of code are run.

Syntax
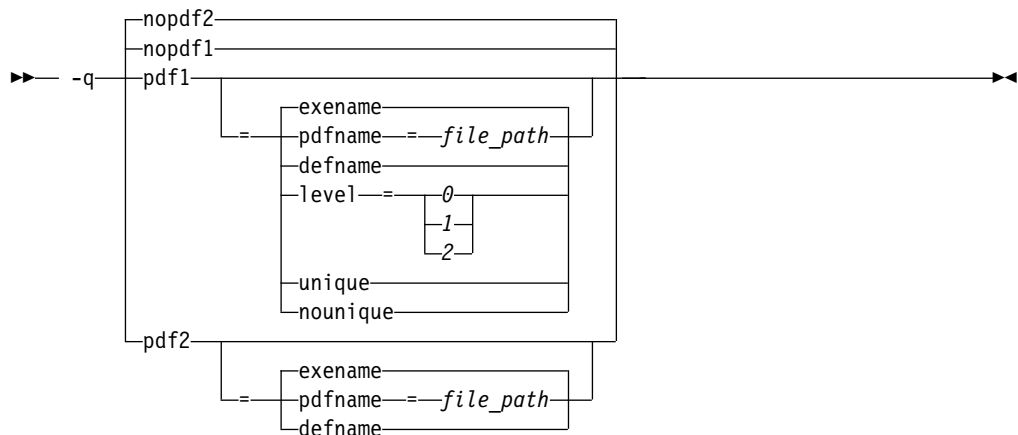
Defaults

-qnopdf1 when -qpdf1 is not specified; -qnopdf2 when -qpdf2 is not specified.

-qpdf1=exename when -qpdf1 is specified without a suboption; -qpdf2=exename when -qpdf2 is specified without a suboption.

Parameters

exename

Names the generated PDF file as *.<output_name>_pdf*, where *<output_name>* is the name of the output file that is generated when you compile your program with **-qpdf1**.

pdfname=*file_path*

Specifies the directories and names for the PDF files and any existing PDF map files. If the PDFDIR environment variable is set, the compiler places the PDF and PDF map files in the directory that is specified by PDFDIR; otherwise, the compiler places these files in the current working directory. If the PDFDIR environment variable is set but the specified directory does not exist, the compiler issues a warning message. The name of the PDF map file follows the name of the PDF file if the **-qpdf1=unique** option is not specified. For example, if you specify the **-qpdf1=pdfname=/home/joe/func** option, the generated PDF file is called func, and the PDF map file is called func_map. Both of the files are placed in the /home/joe directory. You can use the **pdfname** suboption to do simultaneous runs of multiple executable applications by using the same directory. This approach is especially useful when you are tuning dynamic libraries with PDF.

defname

Names the generated PDF file as *._pdf*.

level=0 | 1 | 2

Specifies different levels of profiling information to be generated by the resulting application. The following table shows the type of profiling information supported on each level. The plus sign (+) indicates that the profiling type is supported.

Table 22. Profiling type supported on each -qpdf1 level

Profiling type	Level		
	0	1	2
Block-counter profiling	+	+	+
Call-counter profiling	+	+	+
Value profiling		+	+
Cache-miss profiling			+

-qpdf1=level=1 is the default level. It is equivalent to **-qpdf1**. Higher PDF levels profile more optimization opportunities but have a larger overhead.

Notes:
- Only one application that is compiled with the **-qpdf1=level=2** option can be run at a time on a particular processor.
- Cache-miss profiling information has several levels. Accordingly, if you want to gather different levels of cache-miss profiling information, set the PDF_PM_EVENT environment variable to L1MISS, L2MISS, or L3MISS (if applicable). Only one level of cache-miss profiling information can be instrumented at a time. L2 cache-miss profiling is the default level.
- If you want to bind your application to a specified processor for cache-miss profiling, set the PDF_BIND_PROCESSOR environment variable equal to the processor number.

unique | nounique

You can use the **-qpdf1=unique** option to avoid locking a single PDF file when

multiple processes are writing to the same PDF file in the PDF training step. This option specifies whether a unique PDF file is created for each process during run time. The PDF file name is *<pdf_file_name>*.*<pid>*. *<pdf_file_name>* is one of the following names:

- .*<output_name>*_pdf by default.
- The name that is specified by **pdfname** when this suboption is in effect.
- ._pdf when the **defname** suboption takes effect.

<pid> is the ID of the running process in the PDF training step. For example, if you specify the **-qpdf1=unique:pdfname=abc** option, and there are two processes for PDF training with the IDs 12345678 and 87654321, two PDF files abc.12345678 and abc.87654321 are generated.

Note: When **-qpdf1=unique** is specified, multiple PDF files with process IDs as suffixes are generated. You must use the **mergepdf** program to merge all these PDF files into one after the PDF training step.

Usage

The PDF process consists of the following three steps:

1. Compile your program with the **-qpdf1** option and a minimum optimization level of **-O2**. By default, a PDF map file that is named .*<output_name>*_pdf_map and a resulting application are generated.
2. Run the resulting application with a typical data set. Profiling information is written to a PDF file named .*<output_name>*_pdf by default. This step is called the PDF training step.
3. Recompile and link or relink the program with the **-qpdf2** option and the optimization level used with the **-qpdf1** option. The **-qpdf2** process fine-tunes the optimizations according to the profiling information collected when the resulting application is run.

Predefined macros

None.

Examples

Example 1

The example uses the **-qpdf1=level=0** option to reduce possible runtime instrumentation overhead.

1. Compile all the files with **-qpdf1=level=0**.

   ```
   xlc -qpdf1=level=0 -O3 file1.c file2.c file3.c
   ```
2. Run with one set of input data.

   ```
   ./a.out < sample.data
   ```
3. Recompile all the files with **-qpdf2**.

   ```
   xlc -qpdf2 -O3 file1.c file2.c file3.c
   ```

If the sample data is typical, the program can run faster than without the PDF process.

Example 2

The following example uses the **-qpdf1=level=1** option.

1. Compile all the files with **-qpdf1**.

   ```
   xlc -qpdf1 -O3 file1.c file2.c file3.c
   ```

2. Run with one set of input data.

   ```
   ./a.out < sample.data
   ```

3. Recompile all the files with **-qpdf2**.

   ```
   xlc -qpdf2 -O3 file1.c file2.c file3.c
   ```

If the sample data is typical, the program can now run faster than without the PDF process.

Example 3

The following example uses the **-qpdf1=level=2** option to gather cache-miss profiling information.

1. Compile all the files with **-qpdf1=level=2**.

   ```
   xlc -qpdf1=level=2 -O3 file1.c file2.c file3.c
   ```

2. Set PM_EVENT=L2MISS to gather L2 cache-miss profiling information.

   ```
   export PDF_PM_EVENT=L2MISS
   ```

3. Run with one set of input data.

   ```
   ./a.out < sample.data
   ```

4. Recompile all the files with **-qpdf2**.

   ```
   xlc -qpdf2 -O3 file1.c file2.c file3.c
   ```

If the sample data is typical, the program can now run faster than without the PDF process.

Example 4

This example demonstrates the usage of the **-qpdf[1|2]=exename** option.

1. Compile all the files with **-qpdf1=exename**.

   ```
   xlc -qpdf1=exename -O3 -o final file1.c file2.c file3.c
   ```

2. Run executable with sample input data.

   ```
   ./final < typical.data
   ```

3. List the content of the directory.

   ```
   >ls -lrta

   -rw-r--r-- 1 user staff 50 Dec 05 13:18 file1.c
    -rw-r--r-- 1 user staff 50 Dec 05 13:18 file2.c
    -rw-r--r-- 1 user staff 50 Dec 05 13:18 file3.c
    -rwxr-xr-x 1 user staff 12243 Dec 05 17:00 final
    -rwxr-Sr-- 1 user staff 762 Dec 05 17:03 .final_pdf
   ```

4. Recompile all the files with **-qpdf2=exename**.

   ```
   xlc -qpdf2=exename -O3 -o final file1.c file2.c file3.c
   ```

The program is now optimized by using PDF information.

Example 5

The following example demonstrates the usage of the **-qpdf[1|2]=pdfname** option.

1. Compile all the files with **-qpdf1=pdfname**. The static profiling information is recorded in a file that is named final_map.

   ```
   xlc -qpdf1=pdfname=final -O3 file1.c file2.c file3.c
   ```

2. Run the executable file with sample input data. The profiling information is recorded in a file that is named final.

```
./a.out < typical.data
```

3. List the content of the directory.

```
>ls -lrta

-rw-r--r-- 1 user staff 50 Dec 05 13:18 file1.c
 -rw-r--r-- 1 user staff 50 Dec 05 13:18 file2.c
 -rw-r--r-- 1 user staff 50 Dec 05 13:18 file3.c
 -rwxr-xr-x 1 user staff 12243 Dec 05 18:30 a.out
 -rwxr-Sr-- 1 user staff 762 Dec 05 18:32 final
```

4. Recompile all the files with **-qpdf2=pdfname**.

```
xlc -qpdf2=pdfname=final -O3 file1.c file2.c file3.c
```

The program is now optimized by using PDF information.

Example 6

The following example demonstrates the use of the PDF_BIND_PROCESSOR
environment variable.

1. Compile all the files with **-qpdf1=level=1**.

```
xlc -qpdf1=level=1 -O3 file1.c file2.c file3.c
```

2. Set PDF_BIND_PROCESSOR environment variable so that all processes for this
   executable file are run on processor 1.

```
export PDF_BIND_PROCESSOR=1
```

3. Run executable with sample input data.

```
./a.out < sample.data
```

4. Recompile all the files with **-qpdf2**.

```
xlc -qpdf2 -O3 file1.c file2.c file3.c
```

If the sample data is typical, the program can now run faster than without the PDF
process.

Related information
- "-qshowpdf" on page 197
- "-qipa" on page 160
- -qprefetch
- "-qreport" on page 188
- "cleanpdf" on page 299
- "mergepdf" on page 300
- "showpdf" on page 301
- "Runtime environment variables" on page 18
- "Profile-directed feedback" in the *XL C/C++ Optimization and Programming Guide*

-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

```
                                    ┌─:───────────────────────────┐
                                    ▼                             │
              ┌─prefetch─┐    ─────┬─────────────────────────┬────┘
              │          │         │  ┌─noassistthread─┐      │
   ──q────────┤          ├──=──────┼──┤ assistthread─=─┼─┬─SMT─┬──┤
              │          │         │                   │ └─CMP─┘  │
              │          │         │  ┌─noaggressive─┐            │
              │          │      ─=──┼──┤ aggressive───┤            │
              │          │         │                              │
►►── -q───────┤          │      ─=──┴─dscr──=──value───────────────┤    ──►◄
              └─noprefetch┘
```
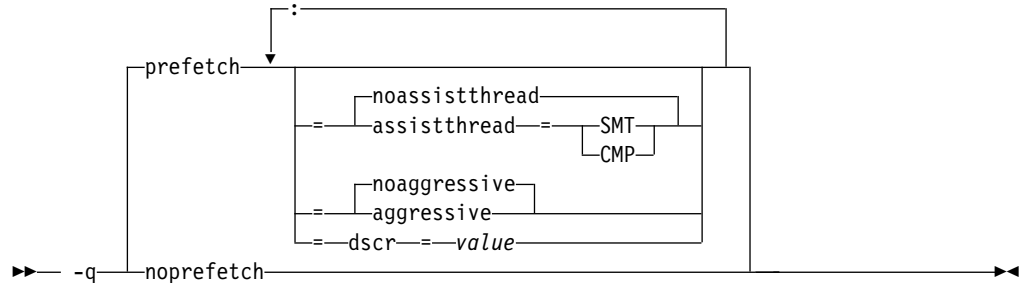
Defaults

-qprefetch=noassistthread:noaggressive:dscr=0

Parameters

assistthread | <u>noassistthread</u>
> When you work with applications that generate a high cache-miss rate, you can use **-qprefetch=assistthread** to exploit assist threads for data prefetching. This suboption guides the compiler to exploit assist threads at optimization level **-O3 -qhot** or higher. If you do not specify **-qprefetch=assistthread**, **-qprefetch=noassistthread** is implied.

> **CMP**
>> For systems based on the chip multi-processor architecture (CMP), you can use **-qprefetch=assistthread=cmp**.

> **SMT**
>> For systems based on the simultaneous multi-threading architecture (SMT), you can use **-qprefetch=assistthread=smt**.

>> **Note:** If you do not specify either CMP or SMT, the compiler uses the default setting based on your system architecture.

aggressive | <u>noaggressive</u>
> This suboption guides the compiler to generate aggressive data prefetching at optimization level **-O3** or higher. If you do not specify **aggressive**, **-qprefetch=noaggressive** is implied.

dscr
> You can specify a value for the dscr suboption to improve the runtime performance of your applications. The compiler sets the Data Stream Control Register (DSCR) to the specified dscr value to control the hardware prefetch engine. The value is valid when **-mcpu=pwr8** is in effect and the optimization level is **-O2** or greater. The default value of dscr is 0.

> *value*

>> The value that you specify for dscr must be 0 or greater, and representable as a 64-bit unsigned integer. Otherwise, the compiler issues a warning message and sets dscr to 0. The compiler accepts both decimal and hexadecimal numbers, and a hexadecimal number requires the prefix of 0x.

The value range depends on your system architecture. See the product information about the POWER Architecture for details. If you specify multiple dscr values, the last one takes effect.

Usage

The **-qnoprefetch** option does not prevent built-in functions such as **__prefetch_by_stream** from generating prefetch instructions.

When you run **-qprefetch=assistthread**, the compiler uses the delinquent load information to perform analysis and generates prefetching assist threads. The delinquent load information can either be provided through the built-in __mem_delay function (const void *delinquent_load_address, const unsigned int delay_cycles), or gathered from dynamic profiling using **-qpdf1=level=2**.

When you use **-qpdf** to call **-qprefetch=assistthread**, you must use the traditional two-step PDF invocation:

1. Run **-qpdf1=level=2**
2. Run **-qpdf2 -qprefetch=assistthread**

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;
}
```

Related information

- -mcpu (-qarch)
- "-qhot" on page 153
- "-qpdf1, -qpdf2" on page 180
- "-qreport" on page 188
- "__mem_delay" on page 528

-qpriority (C++ only)
Category

Object code control

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. You can use the **-qpriority** option to impose a construction order for all static objects declared within the same load module. Destructors for these objects are run in reverse order during termination.

Syntax

Option syntax

►►── -q─priority─=─*number*──►◄

Defaults

The default priority level is 65535.

Parameters

number
>An integer literal in the range of 101 to 65535. A lower value indicates a higher priority; a higher value indicates a lower priority. If you do not specify a *number*, the compiler assumes 65535.

Usage

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.

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 2000, enter the following command:

```
xlc++ myprogram.C -c -qpriority=2000
```

Related information

- "Initializing static objects in libraries" in the *XL C/C++ Optimization and Programming Guide*

-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 that is listed on the command line. When you specify **-qreport** with an option that enables vectorization, the listing file shows a pseudo-C code listing and a summary of how program loops are optimized. The report also includes diagnostic information about why specific loops cannot be vectorized. For example, when **-qreport** is specified with **-qsimd**, messages are provided to identify non-stride-one references that 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. The POWER8 or higher processors support both load and store stream prefetch.

Syntax

```
         ┌─noreport─┐
▶▶── -q──┼─report───┼──────────────────────────────────────────▶◀
```

Defaults

-qnoreport

Usage

To generate a loop transformation listing, you must specify **-qreport** with one of the following options:
- **-qhot**
- **-qsmp**
- **-O3** or higher

You can specify both **-qreport** and **-qpdf2** to generate information in your listing file to help you tune your program. This information is written to the PDF Report section.

To generate a parallel transformation listing or parallel performance messages, you must specify **-qreport** with one of the following options:
- **-qsmp**
- **-O5**
- **-qipa=level=2**

To generate data reorganization information, specify **-qreport** with the optimization level **-qipa=level=2** or **-O5**. Reorganizations include array splitting, array transposing, memory allocation merging, array interleaving, and array coalescing.

To generate information about data prefetch insertion locations, specify **-qreport** with the optimization level of **-qhot** or any other option that implies **-qhot**. This information appears in the LOOP TRANSFORMATION SECTION of the listing file. In addition, when you use **-qprefetch=assistthread** to generate prefetching assist threads, the message: Assist thread for data prefetching was generated also appears in the LOOP TRANSFORMATION SECTION of the listing file.

To generate a list of aggressive loop transformations and parallelization 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 the following command:
```
xlc -qhot -O3 -qreport myprogram.c
```

To compile myprogram.c so the compiler listing also includes a report showing how parallelized loops are transformed, enter the following command:

```
xlc_r -qhot -qsmp -qreport myprogram.c
```

Related information
- "-qhot" on page 153
- "-qsimd" on page 198
- "-qipa" on page 160
- "-qpdf1, -qpdf2" on page 180

-qreserved_reg

Category

Object code control

Pragma equivalent

None.

Purpose

Indicates that the given list of registers cannot be used during the compilation except as a stack pointer, frame pointer or in some other fixed role.

You should use this option in modules that are required to work with other modules that use global register variables or hand-written assembler code.

Syntax

```
►►─── -q─reserved_reg─=─┬─►─register_name─┬───────────────────────►◄
                        └──────: ◄────────┘
```

Defaults

Not applicable.

Parameters

register_name
> A valid register name on the target platform. Valid registers 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:

```
xlc myprogram.c -qreserved_reg=r3:r4
```

-qrestrict

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Specifying this option is equivalent to adding the `restrict` keyword to the pointer parameters within all functions, except that you do not need to modify the source file.

Syntax

```
         ┌─norestrict─┐
►►──-q──┬─          ─┬───────────────────────────────────►◄
         └─restrict───┘
```

Defaults

-qnorestrict. It means no function pointer parameters are restricted, unless you specify the **restrict** attribute in the source file.

Usage

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.

Note: If you specify both the **-qalias=norestrict** and **-qrestrict** options, **-qalias=norestrict** takes effect.

Predefined macros

None.

Examples

To compile `myprogram.c`, instructing the compiler to restrict the pointer access, enter:

```
xlc -qrestrict myprogram.c
```

Related information

- "-fstrict-aliasing (-qalias=ansi), -qalias" on page 108

-qro

Category

Object code control

Purpose

Specifies the storage type for string literals.

When **-qro** or **#pragma strings (readonly)** is in effect, strings are placed in read-only memory. When **-qnoro** or **#pragma strings (writeable)** is in effect, strings are placed in read-write memory.

Syntax

Option syntax

```
►►─ -q─┬─ro──┬──────────────────────────────────────►◄
        └─noro─┘
```

Pragma syntax

```
►►─#─pragma─strings─(─┬─readonly──┬─)────────────────►◄
                      └─writeable─┘
```

Defaults

▶ **C** Strings are read-only for all invocation commands except **cc**. If the **cc** 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:

```
xlc myprogram.c -qnoro
```

Related information

- "-qro" on page 192
- "-qroconst"

-qroconst

Category

Object code control

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

```
►►─ -q─┬─roconst──┬────────────────────────────►◄
        └─noroconst─┘
```

Defaults

- ▶ `C` **-qroconst** for all compiler invocations except **cc** and its derivatives. **-qnoroconst** for the **cc** 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 192

-qrtti, -fno-rtti (-qnortti) (C++ only)
Category

Object code control

Purpose

Generates runtime type identification (RTTI) information for exception handling and for use by the `typeid` and `dynamic_cast` operators.

Syntax

```
►►── -q──┬─rtti──┬──────────────────────────────────────────────►◄
         └─nortti─┘
```

```
►►── -f──no-rtti────────────────────────────────────────────────►◄
```

Defaults

-qrtti

Usage

For improved runtime performance, suppress RTTI information generation with the **-fno-rtti (-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, **-fno-rtti (-qnortti)** specified for others), you may get an undefined symbol error.

Predefined macros
- __GXX_RTTI is predefined to a value of 1 when **-qrtti** is in effect; otherwise, it is undefined.
- __NO_RTTI__ is defined to 1 when **-fno-rtti (-qnortti)** is in effect; otherwise, it is undefined.
- __RTTI_ALL__ is defined to 1 when **-qrtti** is in effect; otherwise, it is undefined.

Related information
- "-fexceptions (-qeh) (C++ only)" on page 98

-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

```
         ┌─nosaveopt─┐
►►── -q───┴─saveopt──┴────────────────────────────────────────────►◄
```

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:

```
                  ┌─f─┐
►►──@(#)──opt──────┼─c─┼───invocation──options───────────────────►◄
                  └─C─┘
```

```
►►──@(#)──cfg──────config_file_options_list──────────────────────►◄
```

```
►►──@(#)──env──────env_var_definition─────────────────────────────►◄
```

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, **xlc**.
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 64,000 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:
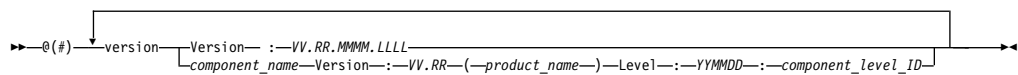
```
►►─@(#)─┬─version─┬─Version─ :─VV.RR.MMMM.LLLL──────────────────────────────────────┬─►◄
         │        └─component_name─Version─:─VV.RR─(─product_name─)─Level─:─YYMMDD─:─component_level_ID─┘
```

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.

component_level_ID

Represents the ID associated with the level of the installed component.

If you want to simply output this information to standard output without writing it to the object file, use the **--version (-qversion)** option.

Predefined macros

None.

Examples

Compile `t.c` with the following command:

```
xlc t.c -c -qsaveopt -qhot
```

Issuing the **strings -a** command on the resulting `t.o` object file produces information similar to the following:

```
IBM XL C/C++ for Linux, Version 13.1.6
@(#)opt c /opt/ibm/xlC/13.1.6/bin/xlC \
-F/opt/ibm/xlC/13.1.6/etc/xlc.cfg.rhel.7.3.gcc.4.8.5 t.c -qhot -qsaveopt -c
@(#)cfg -qalias=ansi -qnostaticlink=libgcc -qthreaded -D_REENTRANT -D__VACPP_MULTI__
```

```
-Wl --no-toc-optimize -qtls -q64 -D_CALL_SYSV -D__null=0
-D__NO_MATH_INLINES -D_CALL_ELF=2 -Wno-parentheses -Wno-unused-value -qtls
@(#)version IBM XL C/C++ for Linux, V13.1.6 (5725-C73, 5765-J08)
@(#)version Version: 13.01.0006.0000
@(#)version Driver Version: 13.1.6(C/C++) Level: 151105 ID: _JbNFoYQ_EeWg_O7EssfHAg
@(#)version C/C++ Front End Version: 13.1.6(C/C++) Level: 151106 ID: _JX7IIIQ_EeWg_O7EssfHAg
@(#)version High-Level Optimizer Version: 13.1.6(C/C++) and 15.1.6(Fortran) Level: 151106
ID: _JfAAgYQ_EeWg_O7EssfHAg
@(#)version Low-Level Optimizer Version: 13.1.6(C/C++) and 15.1.6(Fortran) Level: 151030
ID: _sk208X8mEeWg_O7EssfHAg
```

In the first line, c identifies the source used as C, /opt/ibm/xlC/13.1.6/bin/xlc
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

- "--version (-qversion)" on page 68

-qshowpdf

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

When used with **-qpdf1** and a minimum optimization level of **-02** at compile and
link steps, creates a PDF map file that contains additional profiling information for
all procedures in your application.

Syntax

```
          ┌─showpdf───┐
►►── -q────┼─────────────┼──────────────────────────────────────────────►◄
          └─noshowpdf─┘
```

Defaults

-qshowpdf

Usage

After you run your application with typical data, the profiling information is
recorded into a profile-directed feedback (PDF) file. By default, this PDF file is
named **.**<*output_name*>_pdf, where <*output_name*> is the name of the output file
that is generated when you compile your program with **-qpdf1**.

In addition to the PDF file, the compiler also generates a PDF map file that
contains static information during the PDF1 step. By default, this PDF map file is
named **.**<*output_name*>_pdf_map. With these two files, you can use the **showpdf**

utility to view part of the profiling information of your application in text or XML format. For details of the **showpdf** utility, see "showpdf" in the *XL C/C++ Optimization and Programming Guide*.

If you do not need to view the profiling information, specify the **-qnoshowpdf** option during the PDF1 step so that the PDF map file is not generated. This approach can reduce your compile time.

Predefined macros

None.

Related information
- "-qpdf1, -qpdf2" on page 180
- "showpdf" on page 301
- "Profile-directed feedback" in the *XL C/C++ Optimization and Programming Guide*

-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

```
                          ┌─auto──┐
►►──-q──simd──=──┼─────────┼───────────────────────────────────────────►◄
                          └─noauto─┘
```

Defaults

Whether **-qsimd** is specified or not, **-qsimd=auto** is implied at the **-O3** or higher optimization level; **-qsimd=noauto** is implied at the **-O2** or lower optimization level.

Usage

The **-qsimd=auto** option enables automatic generation of vector instructions for processors that support them. 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. These options are useful for applications with significant image processing demands.

The **-qsimd=noauto** option disables the conversion of loop array operations into vector instructions. To achieve finer control, use **-qstrict=ieeefp**, **-qstrict=operationprecision**, and **-qstrict=vectorprecision**. For details, see "-qstrict" on page 208.

Notes:
- Specifying **-qsimd** without any suboption is equivalent to **-qsimd=auto**.
- Specifying **-qsimd=auto** does not guarantee that autosimdization will occur.
- 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

If you enable IPA and specify **-qsimd=auto** at the IPA compile step, but specify **-qsimd=noauto** at the IPA link step, the compiler automatically sets **-qsimd=auto** at the IPA link step. Similarly, if you enable IPA and specify **-qsimd=noauto** at the IPA compile step, but specify **-qsimd=auto** at the IPA link step, the compiler automatically sets **-qsimd=auto** at the compile step.

Predefined macros

None.

Examples

Any of the following command combinations can enable autosimdization:
- **xlc -O3 -qsimd**
- **xlc -O2 -qhot=level=0 -qsimd=auto**

The following command combination does not enable autosimdization because neither **-O3** nor **-qhot** is specified:
- **xlc -O2 -qsimd=auto**

In the following example, #pragma nosimd is used to disable **-qsimd=auto** for a specific for loop:

```
...
#pragma nosimd
for (i=1; i<1000; i++) {
    /* program code */
}
```

Related information
- "#pragma nosimd" on page 244
- "-mcpu (-qarch)" on page 131
- "-qreport" on page 188
- "-qstrict" on page 208
- *Using interprocedural analysis* in the *XL C/C++ Optimization and Programming Guide*.

-qslmtags
Category

Listings, messages, and compiler information

Pragma equivalent

None.

Purpose

Controls whether SLM Tags logging tracks compiler license usage.

Syntax

```
►►─ -q──┬─noslmtags─┬─────────────────────────────────────────────◄◄
        └─slmtags───┘
```

Defaults

`-qnoslmtags`

Usage

You can specify **-qslmtags** to enable license usage tracking. When **-qslmtags** is in effect, the compiler logs compiler license usage in the SLM Tags format, to a location that you can define by specifying the **slm_dir** attribute of the configuration file. The default location is /var/opt/ibm/xl-compiler/ for a default installation, or $*prefix*/var/opt/ibm/xl-compiler/ for a nondefault installation, where $*prefix* is the nondefault installation path. If you change the default of **slm_dir**, you must create the target directory and set its permissions to be readable, writable, and executable by all compiler users; for example, you can run the following command:

```
chmod 777 $slm_dir
```

The compiler logs each compiler invocation as either a concurrent user or an authorized user invocation, depending on the presence of your uid in a file that lists the authorized users.

Predefined macros

None.

Related information
- Chapter 3, "Tracking compiler license usage," on page 47
- "Configuration file attributes" on page 43

-qsmallstack
Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Minimizes stack usage where possible. Disables optimizations that increase the size of the stack frame.

Syntax

```
►►── -q──┬─nosmallstack─┬──────────────────────────────────────────►◄
         └─smallstack───┘
```

Defaults

-qnosmallstack

Usage

Programs that allocate large amounts of data to the stack, such as threaded programs, might result in stack overflows. The **-qsmallstack** option helps avoid stack overflows by disabling optimizations that increase the size of the stack frame.

This option takes effect only when used together with IPA (the **-qipa**, **-04**, or **-05** compiler options).

Specifying this option might adversely affect program performance.

Predefined macros

None.

Examples

To compile myprogram.c to use a small stack frame, enter the command:
```
xlc myprogram.c -qipa -qsmallstack
```

Related information
- "-g" on page 120
- "-qipa" on page 160
- "-O, -qoptimize" on page 80

-qsmp

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Enables parallelization of program code.

Syntax
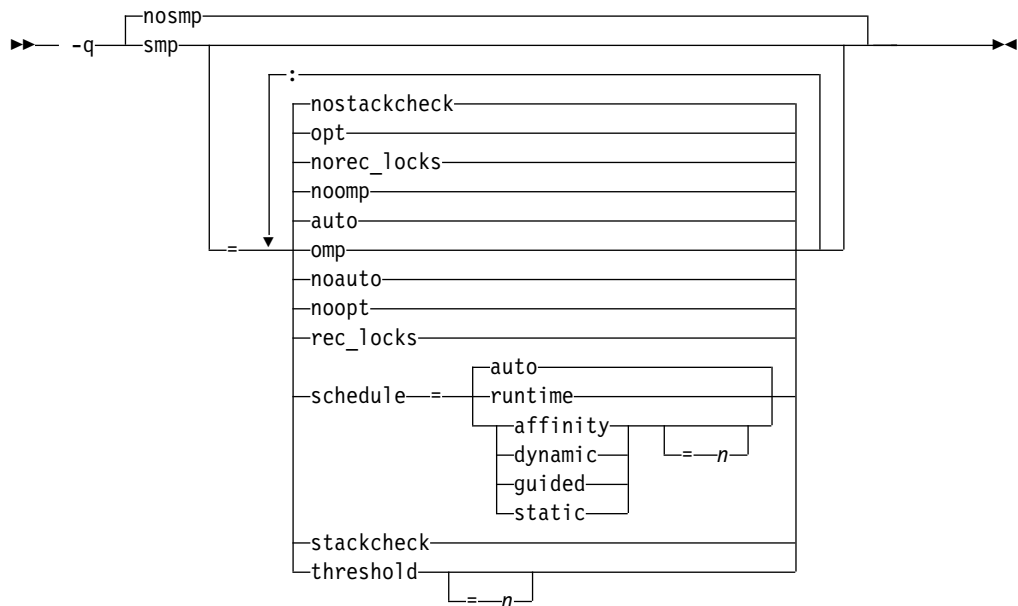
Defaults

-qnosmp. Code is produced for a uniprocessor machine.

Parameters

auto | **noauto**

    auto enables automatic parallelization and optimization of program code; that is the compiler attempts to automatically parallelize both user and compiler-generated loops. **noauto** parallelizes only program code that is explicitly annotated with OpenMP directives. **noauto** is implied if you specify **-qsmp=omp** or **-qsmp=noopt**.

omp | **noomp**

    omp implies **noauto**, that is, only program code that is explicitly annotated with OpenMP directives is parallelized. When **noomp** is in effect, **auto** is implied.

opt | **noopt**

    opt enables optimization of parallelized program code. **noopt** performs the smallest amount of optimization that is required to parallelize the code. This is useful for debugging because **-qsmp** enables the **-O2** and **-qhot** options by default, which may result in the movement of some variables into registers that are inaccessible to the debugger. However, if the **-qsmp=noopt** and **-g** options are specified, these variables will remain visible to the debugger.

rec_locks | **norec_locks**

    Determines whether recursive locks are used. When **rec_locks** is in effect, nested critical sections will not cause a deadlock. Note that the **rec_locks** suboption specifies behavior for critical constructs that is inconsistent with the OpenMP API.

schedule

    Specifies the type of scheduling algorithms and, except in the case of **auto**,

chunk size (*n*) that are used for loops to which no other scheduling algorithm has been explicitly assigned in the source code. Suboptions of the **schedule** suboption are as follows:

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 is not part of the OpenMP API specification.
>
> **Note:** This suboption has been deprecated. You can use the **OMP_SCHEDULE** environment variable with the **dynamic** clause for a similar functionality.

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 that contain *n* iterations each. If *n* is not specified, each chunk contains one iteration.
>
> 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.

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 **floor**(*number_of_iterations*/ *number_of_threads*) iterations. The first **remainder** (*number_of_iterations*/ *number_of_threads*) chunks have one more iteration. Each thread is assigned a separate chunk. 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**.

stackcheck | <u>nostackcheck</u>

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 20.

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:opt:noomp:norec_locks:schedule=auto:
nostackcheck:threshold=100
```

Usage

- Specifying the **omp** suboption always implies **noauto**. Specify **-qsmp=omp:auto** to apply automatic parallelization on OpenMP-compliant applications, as well.
- 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 **-02**. The **-qsmp** option overrides **-qnooptimize**, but does not override **-03**, **-04**, or **-05**. 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 -03** is equivalent to **-qsmp=noopt**, while **-qsmp=noopt -03 -qsmp** is equivalent to **-qsmp -03**.

Related information

- "-O, -qoptimize" on page 80

-qspill

Category

Compiler customization

Pragma equivalent

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

▶▶── -q─spill──=─*size*──◀◀

Defaults

-qspill=512

Parameters

size
> An integer representing the number of bytes for the register allocation spill area.

Usage

If your program is very complex, or if there are too many computations to hold in registers at one time and your program needs temporary storage, you might need to increase this area. Do not enlarge the spill area unless the compiler issues a message requesting a larger spill area. In case of a conflict, the largest spill area specified is used.

Predefined macros

None.

Examples

If you received a warning message when compiling `myprogram.c` and want to compile it specifying a spill area of 900 entries, enter:

```
xlc myprogram.c -qspill=900
```

-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

```
            ┌─nostaticinline─┐
►►── -q─────┴─staticinline───┴──────────────────────────────────────────►◄
```

Defaults

-qnostaticinline

Usage

When **-qnostaticinline** is in effect, any redundant functions definitions for which no bodies are generated are discarded by default.

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() {/*...*/};
```

-qstdinc, -qnostdinc (-nostdinc, -nostdinc++)
Category

Input control

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/ibm/xlC/13.1.6/include/) or by the **-isystem (-qc_stdinc)** option

- ▶ C++ The directory specified in the configuration file for the XL C and C++ header files (this is normally /opt/ibm/xlC/13.1.6/include/) or by the **-isystem (-qcpp_stdinc)** option
- The directory specified in the configuration file for the system header files or by the **-isystem** (**-qgcc_c_stdinc** or **-qgcc_cpp_stdinc**) option

When **-nostdinc++** or **-nostdinc** (**-qnostdinc**) is in effect, these directories are excluded from the search paths. The following directories are searched:

- Directories in which source files containing #include "*filename*" directives are located
- Directories specified by the **-I** option
- Directories specified by the **-include (-qinclude)** option

Syntax

```
►►──┬─-nostdinc++─┬─────────────────────────────────────────►◄
    └─-nostdinc───┘
```

```
            ┌─stdinc───┐
►►── -q──┴─nostdinc─┴──────────────────────────────────────►◄
```
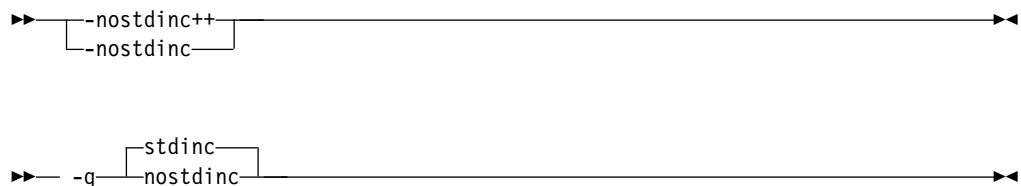
Defaults

-qstdinc

Usage

The search order of header files is described in "Directory search sequence for included files" on page 9.

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:

```
xlc myprogram.c -nostdinc -I/tmp/myfiles
```

Related information
- "-isystem (-qc_stdinc) (C only)" on page 123
- "-isystem (-qcpp_stdinc) (C++ only)" on page 124
- "-isystem (-qgcc_c_stdinc) (C only)" on page 125
- "-isystem (-qgcc_cpp_stdinc) (C++ only)" on page 127

-qstrict

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Ensures that optimizations that are done by default at the **-O3** and higher optimization levels, 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

```
►►── -q ──┬─nostrict──────────────────────────────────────────────────►◄
          └─strict─┐
                   │                    ┌─ : ─────────────────┐
                   │                    │                     │
                   └─ = ─▼─┬─all─────────────────────┐
                           ├─none────────────────────┤
                           ├─precision───────────────┤
                           ├─noprecision─────────────┤
                           ├─exceptions──────────────┤
                           ├─noexceptions────────────┤
                           ├─ieeefp──────────────────┤
                           ├─noieeefp────────────────┤
                           ├─nans────────────────────┤
                           ├─nonans──────────────────┤
                           ├─infinities──────────────┤
                           ├─noinfinities────────────┤
                           ├─subnormals──────────────┤
                           ├─nosubnormals────────────┤
                           ├─zerosigns───────────────┤
                           ├─nozerosigns─────────────┤
                           ├─operationprecision──────┤
                           ├─nooperationprecision────┤
                           ├─vectorprecision─────────┤
                           ├─novectorprecision───────┤
                           ├─order───────────────────┤
                           ├─noorder─────────────────┤
                           ├─association─────────────┤
                           ├─noassociation───────────┤
                           ├─reductionorder──────────┤
                           ├─noreductionorder────────┤
                           ├─guards──────────────────┤
                           ├─noguards────────────────┤
                           ├─library─────────────────┤
                           └─nolibrary───────────────┘
```

Defaults

- **-qstrict** or **-qstrict=all** is always in effect 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 the **-O3** or 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
> Specifying **-qstrict=guards** has the following effects:
> - The compiler does not move operations past guards, which control whether the operations are executed. That is, the compiler does not move operations past guards of the if statements, out of loops, or past guards of function calls that might end the program or throw an exception.
> - When the compiler encounters if expressions that contain pointer wraparound checks that can be resolved at compile time, the compiler does not remove the checks or the enclosed operations. The pointer wraparound check compares two pointers that have the same base but have constant offsets applied to them.
>
> Specifying **-qstrict=noguards** has the following effects:
> - The compiler moves operations past guards.
> - The compiler evaluates if expressions according to language standards, in which pointer wraparounds are undefined. The compiler removes the enclosed operations of the if statements when the evaluation results of the if expressions are false.

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** or **-qfloat=nomaf.**)
- 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 optimization functions enabled by **-qfloat=rsqrt**. You can turn off the optimization functions by using the **-qstrict** option or **-qfloat=norsqrt**. With lower-level or no optimization specified, these optimization functions are turned off by default.

Specifying various suboptions of **-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=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 147.

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 optimization of **-O3** are turned off, and division by the result of a square root is replaced by multiplying by the reciprocal (**-qfloat=rsqrt**), enter:

```
xlc myprogram.c -O3 -qstrict -qfloat=rsqrt
```

To enable all transformations except those affecting precision, specify:

```
xlc myprogram.c -qstrict=none:precision
```

To disable all transformations except those involving NaNs and infinities, specify:

```
xlc myprogram.c -qstrict=all:nonans:noinfinities
```

In the following code example, the `if` expression contains a pointer wraparound check. If you compile the code with the **-qstrict=guards** option in effect, the compiler keeps the enclosed `foo()` function; otherwise, the compiler removes the enclosed `foo()` function.

```
void foo()
{
                // You can add some operations here.
}

int main()
{
  char *p = "a";
  int k = 100;
  if(p + k < p) // This if expression contains a pointer wraparound check.
  {
    foo();      // foo() is the enclosed operation of the if statement.
  }
  return 0;
}
```

Related information
- "-qsimd" on page 198
- "-qfloat" on page 147
- "-qhot" on page 153
- "-O, -qoptimize" on page 80

-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

```
►►─── -q──┬─strict_induction───┬──────────────────────────────────────────────►◄
          └─nostrict_induction─┘
```

Defaults
- **-qstrict_induction**
- **-qnostrict_induction** when **-02** or higher optimization level is in effect

Usage

When using **-02** 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 80

-qtgtarch

Category

Object code control

Pragma equivalent

None.

Purpose

Specifies real or virtual GPU architectures where the code may run. This allows the
compiler to take maximum advantage of the capabilities and machine instructions
that are specific to a GPU architecture, or common to a virtual architecture.

The compiler automatically detects the GPU architecture at compiler configuration
time. The GPU architecture is encoded into the compiler configuration file. You can
override the default by using the **-qtgtarch** option.

Syntax

```
                              ┌─:──────────────────────────┐
                              │    ┌─default──────────────┐ │
►►── -qtgtarch─=─────────────┴────┼─auto─────────────────┼─┴──────────────────►◄
                                   ├─virtual_GPU_architecture─┤
                                   └─real_GPU_architecture────┘
```
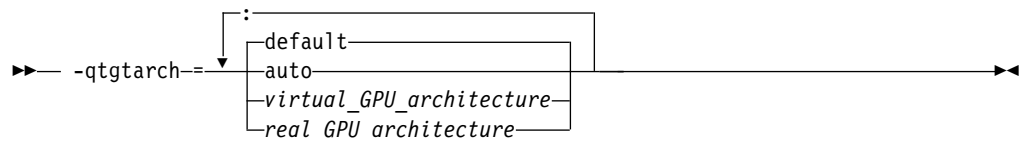
Default

-qtgtarch=default

Parameters

auto
> The architecture of device 0 of the system on which the compiler is being
> executed.

default
> The default architecture, which is determined as follows:
> 1. The architecture specified by the cuda_cc_major and cuda_cc_minor
>   properties which are set in the configuration file;
> 2. If not specified, the architecture of device 0 of the system on which the
>   compiler is being executed;
> 3. If there is no device 0, sm_35.

real_GPU_architecture
> A real GPU architecture, such as sm_35, sm_60, or sm_70, as defined by the CUDA Toolkit.

virtual_GPU_architecture
> A virtual GPU architecture, such as compute_35, compute_60, or compute_70, as defined by the CUDA Toolkit. Virtual GPU architectures specify the features which are supported in the high level PTX code.

Rules

The PTX intermediate code is generated based on the specified virtual GPU architectures and then embedded in the resulting object file or executable. To generate and embed the compiled code images, specify real GPU architectures. The compiled code images for the real GPU architectures are generated from the PTX code.

Each **-qtgtarch** option is used to generate PTX code for exactly one virtual GPU architecture and optionally compiled code images for one or more compatible real GPU architectures. If you need to generate PTX code for multiple virtual GPU architectures, specify the **-qtgtarch** option multiple times, once for each virtual GPU architecture.

The compiler converts between virtual and real GPU architectures when needed, for example, when no virtual architecture is specified, or when multiple virtual GPU architectures are specified.

You can specify the **-qtgtarch** option multiple times, even for the same virtual GPU architecture. The resulting effect is cumulative.

Detailed rules for specifying the **-qtgtarch** option are listed as follows:

*Table 23. Detailed rules for specifying one **-qtgtarch** option*

Number of virtual GPU architectures specified	Number of real GPU architectures specified	The virtual GPU architectures for which the PTX code is generated	The real GPU architectures for which the compiled code images are generated
0	At least one	The virtual GPU architecture corresponding to the lowest level real GPU architecture specified	The real GPU architectures specified
1	0	The virtual GPU architecture specified	N/A **Note:** When no compiled code image is embedded in the resulting object file or executable, a compiled code image will be generated from the PTX code using just-in-time compilation at link or execution time, if needed.

Table 23. Detailed rules for specifying one **-qtgtarch** *option (continued)*

Number of virtual GPU architectures specified	Number of real GPU architectures specified	The virtual GPU architectures for which the PTX code is generated	The real GPU architectures for which the compiled code images are generated
1	At least one	The virtual GPU architecture specified	The real GPU architectures specified
More than one	0	The lowest level virtual GPU architecture specified	The real GPU architectures corresponding to all but the lowest virtual GPU architecture specified
More than one	At least one	The lowest level virtual GPU architecture specified	The real GPU architectures specified and the real GPU architectures corresponding to all but the lowest virtual GPU architecture specified

Predefined macros

None.

Examples

Examples for specifying the **-qtgtarch** option are listed as follows:

Table 24. Examples for specifying the **-qtgtarch** *option*

Command examples	The virtual GPU architectures for which the PTX code is generated	The real GPU architectures for which the compiled code images are generated
`-qtgtarch=sm_60`	compute_60	sm_60 **Note:** The compiled code images are generated from the PTX code.
Assuming the compiler is running on a machine with a GPU with architecture sm_37: `-qtgtarch=auto`	compute_37	sm_37 **Note:** The compiled code image is generated from the PTX code.
`-qtgtarch=compute_35: compute_37:sm_37:sm_60`	compute_35	sm_37 and sm_60 **Note:** The compiled code images are generated from the PTX code.
`-qtgtarch=sm_37:sm_60`	compute_37	sm_37, and sm_60 **Note:** The compiled code images are generated from the PTX code.

Table 24. Examples for specifying the **-qtgtarch** *option (continued)*

Command examples	The virtual GPU architectures for which the PTX code is generated	The real GPU architectures for which the compiled code images are generated
Assuming the compiler is running on a machine with a GPU with architecture sm_37: -qtgtarch=auto:sm_60	compute_37	sm_37, and sm_60 **Note:** The compiled code images are generated from the PTX code.
-qtgtarch=sm_35 -qtgtarch=sm_60	compute_35 and compute_60	sm_35 and sm_60 **Note:** The sm_35 and sm_60 compiled code images are generated from the PTX code for compute_35 and compute_60 correspondingly.

Related information

- *GPU architectures in the CUDA Toolkit documentation,* available at: http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#options-for-steering-gpu-code-generation

-qtimestamps
Category

"Output control" on page 51

Pragma equivalent

None.

Purpose

Controls whether or not implicit time stamps are inserted into an object file.

Syntax

```
►►─── -q──┬─timestamps───┬─────────────────────────────────────────────►◄
          └─notimestamps─┘
```

Defaults

-qtimestamps

Usage

By default, the compiler inserts an implicit time stamp in an object file when it is created. In some cases, comparison tools may not process the information in such binaries properly. Controlling time stamp generation provides a way of avoiding such problems. To omit the time stamp, use the option **-qnotimestamps**.

This option does not affect time stamps inserted by pragmas and other explicit mechanisms.

-qtmplinst (C++ only)

Category

Template control

Pragma equivalent

None.

Purpose

Manages the implicit instantiation of templates.

Syntax

►►── -q─tmplinst──=──none──►◄

Defaults

-qtmplinst=none

Parameters

none
Instructs the compiler to instantiate only inline functions. No other implicit instantiation is performed.

Predefined macros

None.

Related information
• "Explicit instantiation" in the *XL C/C++ Optimization and Programming Guide*

-qxlcompatmacros

Category

Portability and migration

Pragma equivalent

None

Purpose

The option **-qxlcompatmacros** controls definition of the following legacy macros: __xlC__, __xlC_ver__, ► **C++** __IBMCPP__ **C++** ◄ , ► **C** __IBMC__, and __xlc__ **C** ◄ .

For releases starting from IBM XL C/C++ for Linux, V13.1.6, the default is **-qnoxlcompatmacros**. However, compared with earlier versions, this may have impacts on configuring make files with IBM XL C/C++ for Linux for little endian distributions, as both __clang__ and __gcc__ macros are also defined.

You may need to use the **-qxlcompatmacros** option under these circumstances:
- when you migrate programs from IBM XL C/C++ for AIX® to IBM XL C/C++ for Linux for little endian distributions;
- when you migrate programs from IBM XL C/C++ for Linux for big endian distributions to IBM XL C/C++ for Linux for little endian distributions;
- when you migrate programs from earlier versions of XL C/C++ for little endian distributions to the latest.

Alternatively, you can change instances of the legacy macros to `__ibmxl__`, which is always defined in all versions of XL C/C++ for Linux for little endian distributions.

Syntax

```
►►─ -q ─┬─noxlcompatmacros─┬────────────────────────────────────────►◄
        └─xlcompatmacros───┘
```

Defaults

-qnoxlcompatmacros

Predefined macros

The following macros are defined when the **-qxlcompatmacros** option is in effect; otherwise, they are undefined.
- `C++` `__IBMCPP__` `C++`
- `C` `__IBMC__` `C`
- `C` `__xlc__` `C`
- `__xlC__`
- `__xlC_ver__`

Related information
"Macros to identify the XL C/C++ compiler for cross-compilation" on page 306

"-D" on page 74

-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 nonvolatile register saves and restores.

Syntax

```
►►── -q──┬─unwind───┬──────────────────────────────►◄
         └─nounwind─┘
```

Defaults

-qunwind

Usage

The `setjmp` and `longjmp` families of library functions are safe to use with **-qnounwind**.

▶ **C++** Specifying **-qnounwind** also implies **-fno-exceptions (-qnoeh)**.

Predefined macros

None.

Related information
• "-fexceptions (-qeh) (C++ only)" on page 98

-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

```
►►── -r───────────────────────────────────────────►◄
```

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:

```
xlc myprogram.c myprog2.c -r -o mytest.o
```

-s

Category

Object code control

Pragma equivalent

None.

Purpose

Strips the symbol table, line number information, and relocation information from the output file.

This command is equivalent to the operating system **strip** command.

Syntax

►►── -s ──►◄

Defaults

The symbol table, line number information, and relocation information are included in the output file.

Usage

Specifying **-s** saves space, but limits the usefulness of traditional debug programs when you are generating debugging information using options such as **-g**.

Predefined macros

None.

Related information
- "-g" on page 120

-shared (-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

```
►►──-shared────────────────────────────────────────────────►◄
```

```
►►── -q──mkshrobj───────────────────────────────────────────►◄
```

Defaults

By default, the output object is linked with the runtime libraries and startup routines to create an executable file.

Usage

The compiler automatically exports all global symbols from the shared object unless you specify which symbols to export by using the **--version-script** linker option. ► IBM Symbols that have the hidden or internal visibility attribute are not exported. IBM ◄

Specifying **-shared (-qmkshrobj)** implies **-fPIC (-qpic)**.

You can also use the following related options with **-shared (-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*.

-qstaticlink=xllibs
    When you specify **-qstaticlink=xllibs** and **-qmkshrobj**, both options take effect. The compiler creates a shared object in which all references to the XL libraries are statically linked in.

For detailed information about using **-shared (-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:
```
xlc -shared -o big_lib.so lib_a.o lib_b.o lib_c.o
```

Related information

- "-e" on page 93
- "-qipa" on page 160
- "-o" on page 134
- "-fPIC (-qpic)" on page 102
- "-qpriority (C++ only)" on page 187
- "-fvisibility (-qvisibility)" on page 118
- "Supported GCC pragmas" on page 240
- "-static (-qstaticlink)"

-static (-qstaticlink)
Category

Linking

Pragma equivalent

None.

Purpose

Controls whether static or shared runtime libraries are linked into an application.

Syntax

```
►►──-static──────────────────────────────────────────────►◄
             └─-libgcc─┘
```

```
►►──-shared-libgcc───────────────────────────────────────►◄
```

```
►►──── -q──┬─nostaticlink─┬─────────────────────────────────►◄
           └─staticlink───┘    ┌──────:──────┐
                          └─=──▼──┬─libgcc─┬──┘
                                  └─xllibs─┘
```
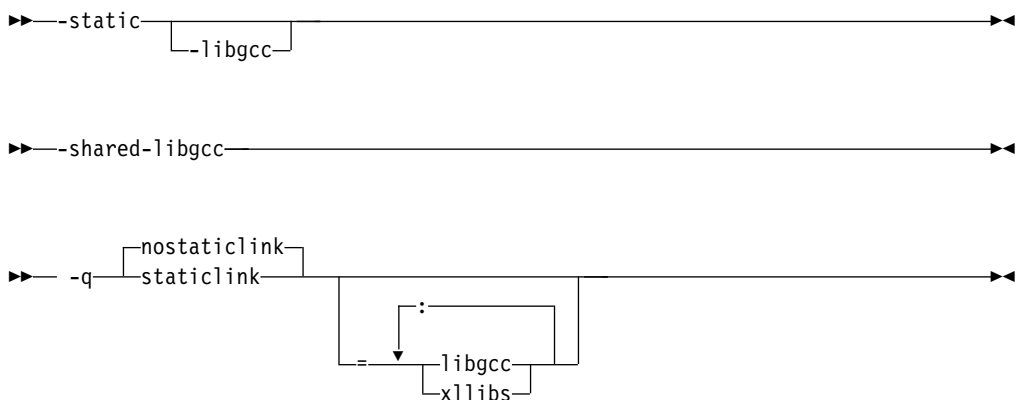
The following table shows the equivalent usage between different format of options for specifying the linkage of shared and nonshared libraries.

Table 25. Option equivalence mapping

Equivalent option	Meaning
-static or -qstaticlink	Build a static object and prevent linking with shared libraries. Every library that is linked to must be a static library.
-shared-libgcc or -qnostaticlink=libgcc	Link with the shared version of libgcc.
-static-libgcc or -qstaticlink=libgcc	Link with the static version of libgcc.

Defaults

-qnostaticlink

Parameters

libgcc

- When you specify **-shared-libgcc**, the compiler links the shared version of **libgcc**.
- When you specify **-static-libgcc**, the compiler links the static version of **libgcc**.

xllibs

- When you specify **xllibs** with **-qnostaticlink**, the compiler links the shared version of the XL compiler libraries.
- When you specify **xllibs** with **-qstaticlink**, the compiler links the static version of the XL compiler libraries.

  The **xllibs** suboption is available only for the **-qstaticlink** and **-qnostaticlink** options.

Usage

When you specify **-static** without suboptions, only static libraries are linked with the object file.

When you specify **-qnostaticlink** without suboptions, shared libraries are linked with the object file.

When you specify **-qstaticlink=xllibs** and **-qmkshrobj**, both options take effect. The compiler links in the static version of XL libraries and creates a shared object at the same time.

When compiler options are combined, conflicts might occur. The following table describes the resolutions of the conflicting compiler options.

Table 26. Examples of conflicting compiler options and resolutions

Options combination examples	Resolution result	Compiler behavior
-qnostaticlink -static-libgcc	Equivalent to **-static-libgcc**	If you first specify **-qnostaticlink** without suboptions and then specify **-static** or **-qstaticlink** with or without suboptions, **-qnostaticlink** is overridden. All libraries are linked statically.
-qnostaticlink -qstaticlink=xllibs	Equivalent to **-qstaticlink=xllibs**	
-static-libgcc -qnostaticlink	Equivalent to **-qnostaticlink**	If you specify **-static** with or without suboptions followed by **-qnostaticlink** without suboptions, **-qnostaticlink** takes effect and shared libraries are linked.
-static -shared-libgcc	Equivalent to **-static**	If you specify **-static** without suboptions followed by **-shared-libgcc** or **-qnostaticlink** with suboptions, **-static** takes effect and only static libraries are linked with the object file.
-static -qnostaticlink=libgcc:xllibs	Equivalent to **-static**	

Table 26. Examples of conflicting compiler options and resolutions (continued)

Options combination examples	Resolution result	Compiler behavior
`-shared-libgcc -static`	Equivalent to **-static**	If you first specify **-shared-libgcc** with suboptions and then specify **-static** without suboptions, **-static** takes effect and all libraries are linked statically.

Notes:

- If a runtime library is linked in statically while its message catalog is not installed on the system, messages are issued with message numbers only, and no message text is shown.
- If a shared library or a dynamically linked application is supposed to throw or catch exceptions, you must link it with the shared **libgcc** by using **-shared-libgcc**.

Predefined macros

None.

Related information
- "-shared (-qmkshrobj)" on page 220

-std (-qlanglvl)
Category

Language element control

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 only)

```
►►─ -q─langlvl─=─┬─extc99────┬──────────────────────────►◄
                 ├─stdc89────┤
                 ├─extc89────┤
                 ├─stdc99────┤
                 ├─extended──┤
                 ├─stdc11────┤
                 └─extc1x────┘
```

-std syntax (C only)

```
>>--std--=--+-gnu9x--------------+------------------------------------><
            +-gnu99--------------+
            +-c89----------------+
            +-c90----------------+
            +-c99----------------+
            +-c9x----------------+
            +-c11----------------+
            +-c1x----------------+
            +-iso9899:1990-------+
            +-iso9899:199409-----+
            +-iso9899:1999-------+
            +-iso9899:199x-------+
            +-iso9899:2011-------+
            +-gnu89--------------+
            +-gnu90--------------+
            +-gnu11--------------+
```

-qlanglvl syntax (C++ only)

```
>>--q--langlvl--=--+-extended---+----------------------------------><
                   +-extended0x-+
                   +-extended1y-+
```
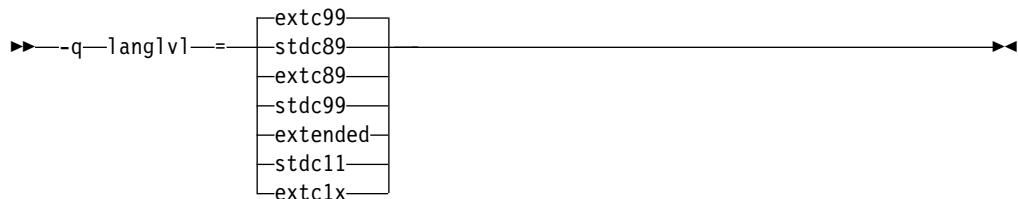
-std syntax (C++ only)
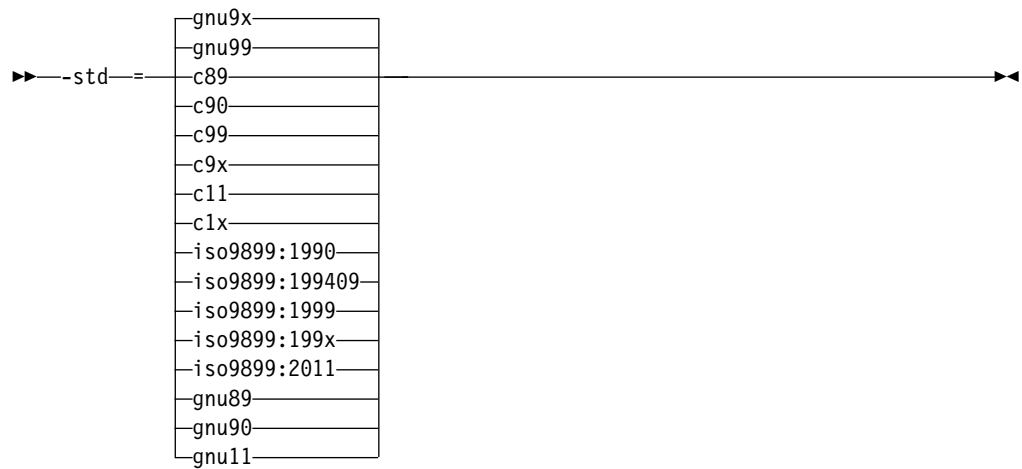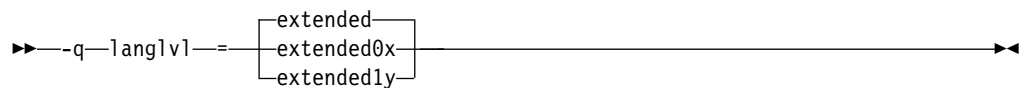
```
>>--std--=--+-gnu++98-+------------------------------------------><
            +-gnu++03-+
            +-c++98---+
            +-c++03---+
            +-c++11---+
            +-gnu++11-+
            +-c++0x---+
            +-gnu++0x-+
            +-c++1y---+
```
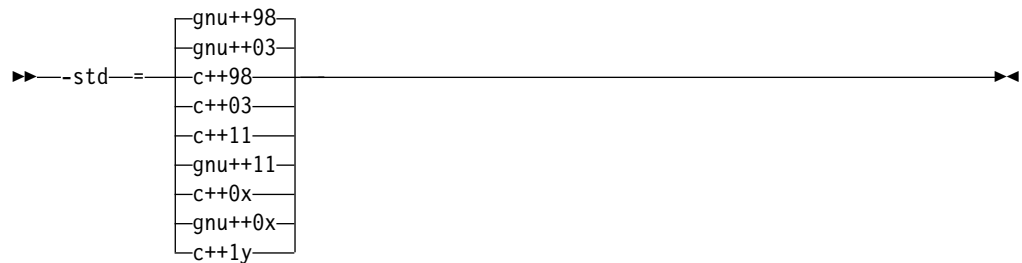
Defaults

- > C **-std=gnu99** or **-std=gnu9x**
- > C++ **-std=gnu++98**
- > C The default is set according to the command used to invoke the compiler:
  - **-qlanglvl=extc99** for the **xlc** and related invocation commands
  - **-qlanglvl=extended** for the **cc** and related invocation commands
  - **-qlanglvl=stdc89** for the **c89** and related invocation commands
  - **-qlanglvl=stdc99** for the **c99** and related invocation commands
- > C++ The default is set according to the command used to invoke the compiler:
  - **-qlanglvl=extended** for the **xlC** or **xlc++** and related invocation commands

Parameters for C language programs

Parameters of the -std option:

c89 | c90 | iso9899:1990

> Compilation conforms strictly to the ANSI C89 standard, also known as ISO C90.

iso9899:199409

> Compilation conforms strictly to the ISO C95 standard.

c99 | c9x | iso9899:1999 | iso9899:199x

> Compilation conforms strictly to the ISO C99 standard, also known as ISO C99.

► C11 **c11 | c1x | iso9899:2011**

> Compilation conforms strictly to the ISO C11 standard. **C11 ◄**

gnu89 | gnu90

> Compilation conforms to the ANSI C89 standard and accepts implementation-specific language extensions, also known as GNU C90.

gnu99 | gnu9x

> Compilation conforms to the ISO C99 standard and accepts implementation-specific language extensions, also known as GNU C99.

gnu11

> Compilation conforms to the ISO C11 standard and accepts implementation-specific language extensions, also known as GNU C11.

If you are using some of the C11 features, you must use the **-qlanglvl** option.

Parameters of the -qlanglvl option:

stdc89

> Compilation conforms strictly to the ANSI C89 standard, also known as ISO C90.

extc89

> Compilation conforms to the ANSI C89 standard and accepts implementation-specific language extensions.

stdc99

> Compilation conforms strictly to the ISO C99 standard.

extc99

> Compilation conforms to the ISO C99 standard and accepts implementation-specific language extensions.

extended

> Compilation is based on the ISO C89 standard, with some differences to accommodate extended language features.

► C11 **stdc11**

> Compilation conforms strictly to the ISO C11 standard. **C11 ◄**

► C11 **extc1x**

> Compilation is based on the C11 standard, invoking all the currently supported C11 features and other implementation-specific language extensions. **C11 ◄**

The following tables reflect the mapping between the **-qlanglvl** and **-std** suboptions:

Table 27. Mapping between the **-qlanglvl** *and* **-std** *suboptions (C only)*

-qlanglvl suboption	Mapping to -std suboption
stdc89	c89	c90	iso9899:1990

-qlanglvl suboption	Mapping to -std suboption
`extc89`	`gnu89` │ `gnu90`
`stdc99`	`c99` │ `c9x` │ `iso9899:1999` │ `iso9899:199x`
`extc99`	`gnu99` │ `gnu9x`
`stdc11`	`c11` │ `c1x` │ `iso9899:2011`
`extc1x`	`gnu11`

Parameters for C++ language programs

Parameters of the `-std` option:

`gnu++98` │ `gnu++03`
>   Compilation is based on the ISO C++98 standard, with some differences to
>   accommodate extended language features.

`c++98` │ `c++03`
>   Compilation conforms strictly to the ISO C++ standard, also known as ISO
>   C++98.

▶ C++11 **`c++11` │ `c++0x`**
>   Compilation conforms strictly to the ISO C++ standard plus amendments, also
>   known as ISO C++11. C++11◀

▶ C++11 **`gnu++11` │ `gnu++0x`**
>   Compilation is based on the ISO C++ standard, with some differences to
>   accommodate extended language features. C++11◀

▶ C++14 **`c++1y`**
>   Compilation is based on the C++14 standard, invoking most of the C++11
>   features and all the currently supported C++14 features. C++14◀

Parameters of the `-qlanglvl` option:

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

▶ C++11 **`extended0x`**
>   Compilation is based on the C++11 standard, invoking most of the C++
>   features and all the currently-supported C++11 features. C++11◀

▶ C++14 **`extended1y`**
>   Compilation is based on the C++14 standard, invoking most of the C++11
>   features and all the currently supported C++14 features.

>   **Note:** IBM supports selected features of C++14 standard. IBM will continue to
>   develop and implement the features of this standard. The implementation of
>   the language level is based on IBM's interpretation of the standard. Until IBM's
>   implementation of all the C++14 features is complete, including the support of
>   a new C++14 standard library, the implementation might 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 C++14 features.
>   C++14◀

The following tables reflect the mapping between the **-qlanglvl** and **-std** suboptions:

Table 28. Mapping between the **-qlanglvl** *and* **-std** *suboptions (C++ only)*

-qlanglvl suboption	Mapping to -std suboption
extended	gnu++98 | gnu++03
extended0x	gnu++11 | gnu++0x
extended1y	c++1y

Predefined macros

See "Macros related to language levels" on page 311 for a list of macros that are predefined by **-qlanglvl** suboptions.

-t

Category

Compiler customization

Pragma equivalent

None.

Purpose

Applies the prefix specified by the **-B** option to the designated components.

Syntax
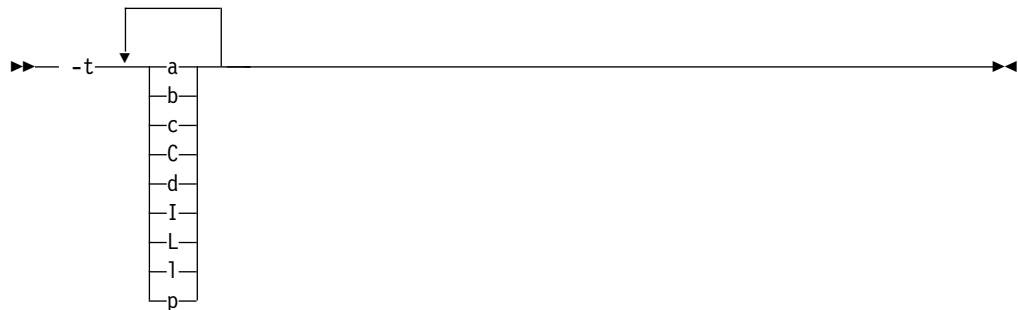
Defaults

The default paths for all of the compiler components are defined in the compiler configuration file.

Parameters

The following table shows the correspondence between **-t** parameters and the component names:

Parameter	Description	Component name
a	The assembler	as

Parameter	Description	Component name
b	The low-level optimizer	xlCcode
c, C	The C and C++ compiler front end	xlCentry
d	The disassembler	dis
I (uppercase i)	The high-level optimizer, compile step	ipa
L	The high-level optimizer, link step	ipa
l (lowercase L)	The linker	ld
p	The preprocessor	xlCentry

Usage

Use this option 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:

```
xlc myprogram.c -B/u/newones/compilers/ -tca
```

Related information
- "-B" on page 72

-v, -V

Category

Listings, messages, and compiler information

Pragma equivalent

None.

Purpose

Reports the progress of compilation, by naming the programs being invoked and the options being specified to each program.

When the **-v** option is in effect, information is displayed in a comma-separated list. When the **-V** option is in effect, information is displayed in a space-separated list.

Syntax

►►──┬─ -v ─┬──►◄
　　 └─ -V ─┘

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:

```
xlc myprogram.c -v
```

Related information
* "-### (-#) (pound sign)" on page 66

-w

Category

Listings, messages, and compiler information

Pragma equivalent

None.

Purpose

Suppresses warning messages.

Syntax

►►──── -w ──►◄

Defaults

All informational and warning messages are reported.

Usage

Informational and warning messages that supply additional information to a severe error are not disabled by this option.

Predefined macros

None.

Examples

Consider the file `myprogram.c`.

```
#include <stdio.h>
int main()
 { char* greeting = "hello world";
   printf("%d \n", greeting);
    return 0;
}
```

- If you compile `myprogram.c` without the **-w** option, the compiler issues a warning message.

  ```
  xlC myprogram.c
  ```

  Output:

  ```
  "5:18: warning: format specifies type 'int' but the argument has type 'char *' [-Wformat]
  printf("%d \n", greeting);
  ~~ ~~~~~~
  %s
  1 warning generated."
  ```

- If you compile `myprogram.c` with the **-w** option, the warning message is suppressed.

  ```
  xlC myprogram.c -w
  ```

-x (-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 **-x** option instructs the compiler not to rely on the file name suffix, and to instead assume a source type as specified by the option.

Syntax

```
                ┌─none────────────────┐
►►── -x──────┼─assembler───────────┼──────────────────────────────────►◄
                ├─assembler-with-cpp──┤
                ├─c───────────────────┤
                └─c++─────────────────┘
```

```
         ┌─default─────────┐
►►─ -q─sourcetype─=─┼─assembler────────┼──────────────────────────────────►◄
                   ├─assembler-with-cpp─┤
                   ├─c────────────────┤
                   └─c++──────────────┘
```
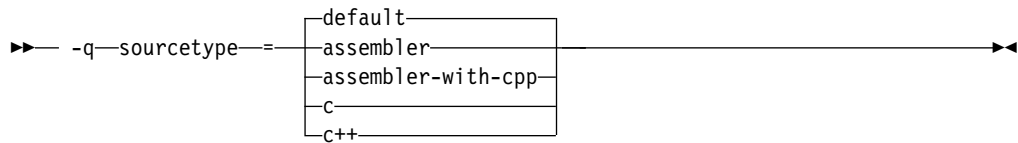
Defaults

-x none or **-qsourcetype=default**

Parameters

assembler
> All source files following the option are compiled as if they are assembler language source files.

assembler-with-cpp
> All source files following the option are compiled as if they are assembler language source files that need preprocessing.

c All source files following the option are compiled as if they are C language source files.

c++
> All source files following the option are compiled as if they are C++ language source files. This suboption is equivalent to the **-+** option.

default (-qsourcetype only)
> The programming language of a source file is implied by its file name suffix.

none (-x only)
> 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.

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:

```
xlc goodbye.C -x c hello.C
```

hello.C is compiled as a C source file, but goodbye.C is compiled as a C++ file.

Predefined macros

None.

Related information
* "-+ (plus sign) (C++ only)" on page 67

-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

```
►►─── -y─┬─n─┬──────────────────────────────────────────── ◄◄
         ├─m─┤
         ├─p─┤
         └─z─┘
```
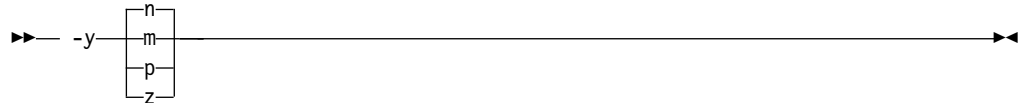
Defaults

- **-yn**

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:

```
xlc myprogram.c -yz
```

Supported GCC options

The following GCC options are also supported in IBM XL C/C++ for Linux, V13.1.6. For details about these options, see the GNU Compiler Collection online documentation at http://gcc.gnu.org/onlinedocs/.

- @file
- -###
- --help
- --sysroot

- --version
- -ansi
- -dD
- -dM
- -fansi-escape-codes
- -fasm, -fno-asm
- -fcolor-diagnostics
- -fcommon, -fno-common
- -fconstexpr-depth
- -fconstexpr-steps
- -ffast-math
- -fdiagnostic-parsable-fixits
- -fdiagnostic-show-category=[none|id|name]
- -fdiagnostic-show-template-tree
- -fdiagnostics-fixit-info
- -fdiagnostics-format=[clang|msvc|vi]
- -fdiagnostics-print-source-range-info
- -fdiagnostics-show-name
- -fdiagnostics-show-option
- -fdollars-in-identifiers, -fno-dollars-in-identifiers
- -fdump-class-hierarchy
- -fexceptions, -fno-exceptions
- -ffreestanding
- -fgnu89-inline
- -fhosted
- -finline-functions
- -fmessage-length
- -fno-access-control
- -fno-assume-sane-operator-new
- -fno-builtin
- -fno-diagnostics-show-caret
- -fno-diagnostics-show-option
- -fno-elide-type
- -fno-gnu-keywords
- -fno-operator-names
- -fno-rtti
- -fno-show-column
- -fpack-struct
- -fpermissive
- -fPIC, -fno-PIC
- -fPIE, -fno-PIE
- -fsemantic-interposition, -fno-semantic-interposition
- -fshort-enums
- -fshort-wchar
- -fshow-column

- -fshow-source-location
- -fsigned-char, -fno-signed-char
- -fstrict-aliasing
- -fsyntax-only
- -ftabstop=*width*
- -ftemplate-backtrace-limit
- -ftemplate-depth
- -ftime-report
- -ftls-model, -fno-tls-model
- -ftrap-function=*name*
- -ftrapping-math, -fnotrapping-math
- -funsigned-char, -fno-unsigned-char
- -funroll-all-loops
- -funroll-loops
- -fvisibility
- -gsplit-dwarf [1]
- -idirafter
- -imacros
- -include
- -iprefix
- -iquote
- -isysroot
- -isystem
- -iwithprefix
- -maltivec, -mno-altivec
- -maltivec=be
- -maltivec=le
- -mcpu
- -mtune
- -M
- -MD
- -MF
- -MG
- -MM
- -MMD
- -MP
- -MQ
- -MT
- -nodefaultlibs
- -nostartfiles
- -nostdinc
- -nostdinc++
- -pedantic
- -pedantic-errors
- -pie

- -rdynamic
- -shared
- -shared-libgcc
- -static
- -static-libgcc
- -std
- -trigraphs
- -w
- -Wall
- -Wambiguous-member-template
- -Wbad-function-cast
- -Wbind-to-temporary-copy
- -Wc++11-compat
- -Wcast-align
- -Wchar-subscripts
- -Wcomment
- -Wconversion
- -Wdelete-non-virtual-dtor
- -Wempty-body
- -Wenum-compare
- -Werror
- -Werror=foo [specically, -Werror=unused-command-line-argument to switch between warning/error for invalid options]
- -Weverything
- -Wextra-tokens
- -Wfatal-errors
- -Wfloat-equal
- -Wfoo
- -Wformat-nonliteral
- -Wformat-security
- -Wformat-y2k
- -Wignored-qualifiers
- -Wimplicit
- -Wimplicit-function-declaration
- -Wimplicit-int
- -Wmain
- -Wmissing-braces
- -Wmissing-field-initializers
- -Wmissing-prototypes
- -Wnarrowing
- -Wno-attributes
- -Wno-builtin-macro-redefined
- -Wno-deprecated
- -Wno-deprecated-declarations
- -Wno-division-by-zero

- -Wno-endif-labels
- -Wno-extra-tokens
- -Wno-format
- -Wno-format-extra-args
- -Wno-format-zero-length
- -Wno-int-conversion
- -Wno-int-to-pointer-cast
- -Wno-invalid-offsetof
- -Wno-multichar
- -Wnonnull
- -Wno-return-local-addr
- -Wno-unused-result
- -Wno-virtual-move-assign
- -Wnon-virtual-dtor
- -Woverlength-strings
- -Woverloaded-virtual
- -Wpadded
- -Wparantheses
- -Wpedantic
- -Wpointer-arith
- -Wpointer-sign
- -Wreorder
- -Wreturn-type
- -Wsequence-point
- -Wshadow
- -Wsign-compare
- -Wsign-conversion
- -Wsizeof-pointer-memaccess
- -Wstack-protector
- -Wswitch
- -Wsystem-headers
- -Wtautological-compare
- -Wtrigraphs
- -Wtype-limits
- -Wundef
- -Wuninitialized
- -Wunknown-pragmas
- -Wunused
- -Wunused-label
- -Wunused-parameter
- -Wunused-value
- -Wunused-variable
- -Wvarargs
- -Wvariadic-macros
- -Wvla

- -Wwrite-strings
- -x
- -X

Note:

1. ▶ `GPU` If your program targets an NVIDIA GPU device by using the **-qoffload** option, **-gsplit-dwarf** affects the debug information for host code only. This is because the debug information for device code is stored in a special section of the object file or executable file and thus cannot be split out. `GPU` ◀

Chapter 5. Compiler pragmas reference

The following sections describe the available pragmas:
- "Pragma directive syntax"
- "Scope of pragma directives"
- "Supported GCC pragmas" on page 240
- "Supported IBM pragmas" on page 240

Pragma directive syntax

XL C/C++ supports the following forms of pragma directives:

#pragma *name*
> This form uses the following syntax:

```
►►─#─pragma─┬──────────────────────┬─────────────────────────►◄
            └─▼─name─(─suboptions─)─┘
```

> 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:

```
►►──_Pragma─(─"─┬──────────────────────┬─"─)───────────────────►◄
                └─▼─name─(─suboptions─)─┘
```

> 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.

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.

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.

Supported GCC pragmas

The following GCC pragmas are supported in IBM XL C/C++ for Linux, V13.1.6. For details about these pragmas, see the GNU Compiler Collection online documentation at http://gcc.gnu.org/onlinedocs/.

- #pragma GCC dependency
- #pragma GCC diagnostic *kind option*
- #pragma GCC diagnostic pop
- #pragma GCC diagnostic push
- #pragma GCC error *string*
- #pragma GCC poison
- #pragma GCC system_header
- #pragma GCC visibility push(visibility)
- #pragma GCC visibility pop
- #pragma GCC warning *string*
- #pragma message *string*
- #pragma once
- #pragma pop_macro("*macro_name*")
- #pragma push_macro("*macro_name*")
- #pragma redefine_extname *oldname newname*
- #pragma unused

Supported IBM pragmas

This section contains descriptions of individual pragmas available in XL C/C++.

For each pragma, the following information is given:

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 239 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 disjoint

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

```
►►──#pragma disjoint──────────────────────────────────────────────────►

►─(──┬──────┬──variable_name──▼──,──┬──────┬──variable_name──)──────►◄
     │  ┌─┐ │                       │  ┌─┐ │
     └──▼─*─┘                       └──▼─*─┘
```
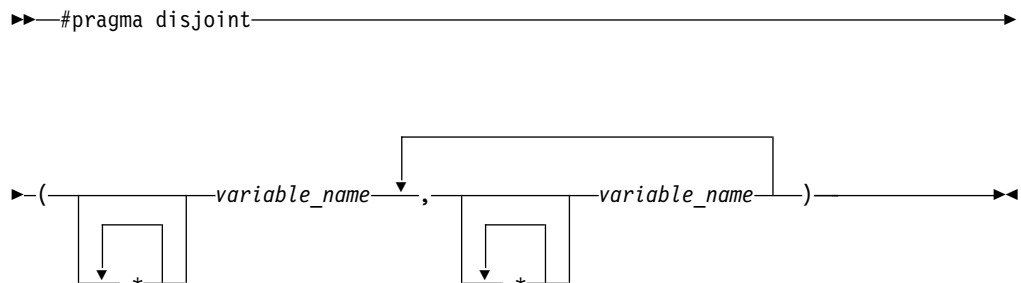
Parameters

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 only in the function or block scope. 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.

Examples

The following example shows the use of **#pragma disjoint**.

```
            int a, b, *ptr_a, *ptr_b;

            one_function()
            {
              #pragma disjoint(*ptr_a, b)   /* *ptr_a never points to b */
              #pragma disjoint(*ptr_b, a)   /* *ptr_b never points to a */

              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 execution_frequency

Purpose

Marks program source code that you expect will be either very frequently or very infrequently executed.

When optimization is enabled, the pragma is used as a hint to the optimizer.

Syntax

```
►►─#─pragma─execution_frequency─(─┬─very_low──┬─)─────────────────────►◄
                                  └─very_high─┘
```

Parameters

very_low
  Marks source code that you expect will be executed very infrequently.

very_high
  Marks source code that you expect will be executed very frequently.

Usage

Use this pragma in conjunction with an optimization option; if optimization is not enabled, the pragma has no effect.

The pragma must be placed within block scope, and acts on the closest preceding 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.    */
```

#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

```
►►─#─pragma─ibm independent_loop─┬──────────┬─────────────────────►◄
                                 └─if exp─┘
```

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 only takes effect if you specify the -qhot compiler option.

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 nosimd
Purpose

Disables automatic generation of vector instructions. This pragma needs to be specified on a per-loop basis.

Syntax

►►—#—pragma—nosimd———————————————————————————————►◄

Example

In the following example, #pragma nosimd is used to disable **-qsimd=auto** for a specific for loop.

```
...
#pragma nosimd
for (i=1; i<1000; i++)
{
    /* program code */
}
```

Related reference:

"-qsimd" on page 198

#pragma option_override
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

```
►►—#—pragma—option_override—————————————————————————————————►

►—(—identifier—,—"—opt—(—level—,——0——)—"—)—)———————————————►◄
                                    ├—2—┤
                                    └—3—┘
```

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[1]
level, 2	-O2[1]
level, 3	-O3[2]

Notes:

1. If optimization level **-O3** or higher is specified on the command line, #pragma option_override(*identifier*, "opt(level, 0)") or #pragma option_override(*identifier*, "opt(level, 2)") does not turn off the implication of the **-qhot** and **-qipa** options.

2. Specifying **-O3** implies **-qhot=level=0**. However, specifying #pragma option_override(*identifier*, "opt(level, 3)") in source code does not imply **-qhot=level=0**.

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 80
- "-qstrict" on page 208

#pragma pack

Purpose

Sets the alignment of all aggregate members to a specified byte boundary.

If the byte boundary number is smaller than the natural alignment of a member, padding bytes are removed, thereby reducing the overall structure or union size.

Syntax
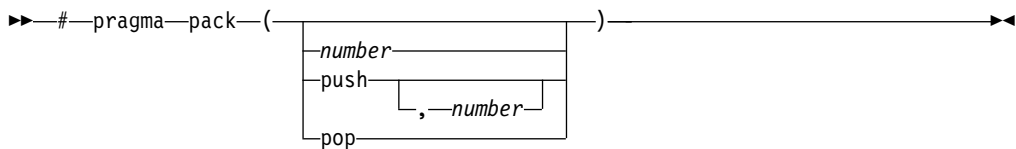
Defaults

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

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.

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.

pop

Removes the previous value added with **#pragma pack**. Specifying **#pragma pack()** with no parameters is equivalent to **#pragma pack(pop)**.

Usage

The **#pragma pack** directive applies to the definition of an aggregate type, rather than to the declaration of an instance of that type; it therefore automatically applies to all variables declared of the specified type.

The **#pragma pack** directive modifies the current alignment rule for only the members of structures whose declarations follow the directive. It does not affect the alignment of the structure directly, but by affecting the alignment of the members of the structure, it may affect the alignment of the overall structure.

The **#pragma pack** directive cannot increase the alignment of a member, but rather can decrease the alignment. For example, for a member with data type of short, a **#pragma pack(1)** directive would cause that member to be packed in the structure on a 1-byte boundary, while a **#pragma pack(4)** directive would have no effect.

The **#pragma pack** directive causes bit fields to cross bit field container boundaries.

```
#pragma pack(2)
struct A{
  int a:31;
  int b:2;
}x;

int main(){
  printf("size of struct A = %lu\n", sizeof(x));
}
```

When the program is compiled and run, the output is:

```
size of struct A = 6
```

But if you remove the **#pragma pack** directive, you get this output:

```
size of struct A = 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:

```
  ┌──── nonpacked[0] ─────┬──── nonpacked[1] ────┐
  │                       │                      │
  │       a               │        a             │
  │   x   │   y   │   z    │    x   │   y   │  z   │
  │       │       │        │        │       │      │
  0   1   2   3   4   5   6   7   8
```
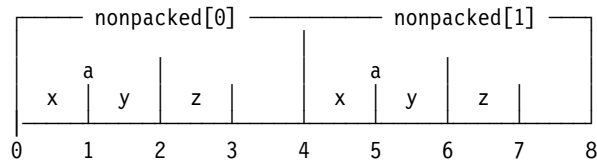
The next example uses **#pragma pack(1)** to set the alignment of unions of type uu to 1 byte:

```
#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:

```
  ┌──── packed[0] ────┬──── packed[1] ───┐
  │                   │                  │
  │      a            │       a          │
  │   x  │  y  │   z   │   x  │  y  │  z  │
  │      │     │       │      │     │     │
  0   1   2   3   4   5   6
```
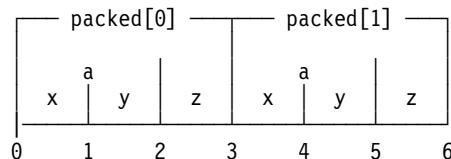
Related information
- "-fpack-struct (-qalign)" on page 103
- "Using alignment modifiers" in the *XL C/C++ Optimization and Programming Guide*

#pragma reachable
Purpose

Informs the compiler that the point in the program after a named function can be the target of a branch from some unknown location.

By informing the compiler that the instruction after the specified function can be reached from a point in your program other than the return statement in the named function, the pragma allows for additional opportunities for optimization.

Note: The compiler automatically inserts **#pragma reachable** directives for the setjmp family of functions (setjmp, _setjmp, sigsetjmp, and _sigsetjmp) when you include the setjmp.h header file.

Syntax

```
►►─#─pragma─reachable─(─┬─────────────┬─function_name─┬─)──────────────────────►◄
                        └──────,──────┘
```

Parameters

function_name
   The name of a function preceding the instruction which is reachable from a point in the program other than the function's return statement.

Defaults

Not applicable.

#pragma simd_level
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

```
►►─#─pragma─simd_level─(─n─)─────────────────────────────────────────────────►◄
```

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 153 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 */

} ...
```

#pragma STDC CX_LIMITED_RANGE
Purpose

Informs the compiler that complex division and absolute value are only invoked with values such that intermediate calculation will not overflow or lose significance.

Syntax

```
►►──#──pragma──STDC cx_limited_range──┬─off─────┬─────────────────────────►◄
                                      ├─on──────┤
                                      └─default─┘
```

Usage

Using values outside the limited range may generate wrong results, where the limited range is defined such that the "obvious symbolic definition" will not overflow or run out of precision.

The pragma is effective from its first occurrence until another **cx_limited_range** pragma is encountered, or until the end of the translation unit. When the pragma occurs inside a compound statement (including within a nested compound statement), it is effective from its first occurrence until another **cx_limited_range** pragma is encountered, or until the end of the compound statement.

Examples

The following example shows the use of the pragma for complex division:

```
#include <complex.h>

_Complex double a, b, c, d;
void p() {

d = b/c;

{

#pragma STDC CX_LIMITED_RANGE ON

a = b / c;

}
}
```

The following example shows the use of the pragma for complex absolute value:

```
#include <complex.h>

_Complex double cd = 10.10 + 10.10*I;
int p() {

#pragma STDC CX_LIMITED_RANGE ON

double d = cabs(cd);
}
```

#pragma unroll, #pragma nounroll
Purpose

Controls loop unrolling, for improved performance.

Syntax

```
►►──#──pragma──┬─nounroll─────────────┬──────────────────►◄
               └─unroll──┬──────────┬─┘
                         └─(──n──)──┘
```

Parameters

n Instructs the compiler to unroll loops by a factor of *n*. In other words, the body
     of a loop is replicated to create *n* copies (including the original) and the
     number of iterations is reduced by a factor of *1/n*. The value of *n* must be a
     positive integer.

     Specifying **#pragma unroll(1)** disables loop unrolling, and is equivalent to
     specifying **#pragma nounroll**.

Usage

Only one pragma can be specified on a loop.

The pragma affects only the loop that follows it. An inner nested loop requires a
#pragma unroll directive to precede it if the wanted loop unrolling strategy is
different from that of the **-funroll-loops (-qunroll)** option.

The **#pragma unroll** and **#pragma nounroll** directives can only be used on `for` loops. 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.

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 reference:

"-funroll-loops (-qunroll), -funroll-all-loops (-qunroll=yes)" on page 116

Pragma directives for OpenMP parallelization

You can use OpenMP pragma directives in your program source to control parallel processing.

The pragmas take effect only when parallelization is enabled with the **-qsmp** compiler option.

#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

Syntax form 1

```
►►──#──pragma──omp atomic──┬──────────┬──┬─────────┬──┬──────────┬──►◄
                           └─seq_cst──┘  │ ┌─update─┐│  └─seq_cst─┘
                                         ├──read────┤
                                         ├──write───┤
                                         └──capture─┘
```

```
►►──expression_statement─────────────────────────────────────────►◄
```

Syntax form 2

```
►►──#──pragma──omp atomic──┬──────────┬──capture──┬──────────┬──►◄
                           └─seq_cst──┘           └─seq_cst──┘
```

```
►►──structured_block──────────────────────────────────────────►◄
```
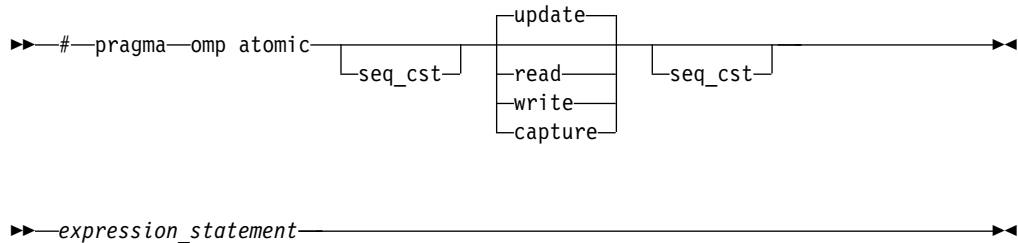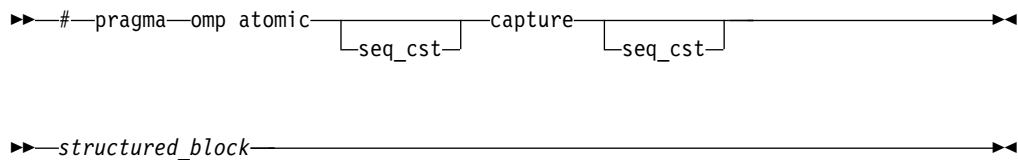
where *expression_statement* is an expression statement of scalar type, and *structured_block* is a structured block of two expression statements.

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.

seq_cst
>   Supports sequentially atomic operations by forcing atomically performed
>   operations to include an implicit flush operation without a list. At most one
>   **seq_cst** clause can be specified for one directive.

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;
	
	x = expr binop x;	

read	v = x;	
write	x = expr;	

capture	v = x++;
	
	v = x--;
	
	v = ++x;
	
	v = --x;
	
	v = x binop = expr;
	
	v = x = x binop expr;
	
	v = x = expr binop x;
		{v = x; x binop = expr;}
		
		{v = x; xOP;}
		
		{v = x; OPx;}
		
		{x binop = expr; v = x;}
		
		{xOP; v = x;}
		
		{OPx; v = x;}
		
		{v = x; x = x binop expr;}
		
		{x = x binop expr; v = x;}
		
		{v = x; x = expr binop x;}
		
		{x = expr binop x; v = x;}
		
		{v = x; x = expr;}[1]

Note: 1. This expression is to support atomic swap operations.		

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.

▶ `GPU`　The **omp atomic** directive is not supported on a target device.　`GPU` ◀

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 barrier
Purpose

The **omp barrier** directive identifies a synchronization point at which threads in a parallel region will not execute beyond the **omp barrier** until all other threads in the team complete all explicit tasks in the region.

Syntax

►►—#—pragma—omp barrier————————————————————————►◄

Usage

The **omp barrier** directive must appear within a block or compound statement. For example:
```
if (x!=0) {
   #pragma omp barrier    /* valid usage    */
}
if (x!=0)
   #pragma omp barrier    /* invalid usage  */
```

#pragma omp critical
Purpose

The **omp critical** directive identifies a section of code that must be executed by a single thread at a time.

Syntax

```
                              ┌─,──────┐
►►—#—pragma—omp critical——────▼─(name)─┘────────────────────►◄
```

where *name* can optionally be used to identify the critical region. Identifiers naming a critical region have external linkage and occupy a namespace distinct from that used by ordinary identifiers.

Usage

A thread waits at the start of a critical region identified by a given name until no other thread in the program is executing a critical region with that same name.

Critical sections not specifically named by **omp critical** directive invocation are mapped to the same unspecified name.

#pragma omp declare target
Purpose

The **omp declare target** directive specifies that variables and functions are mapped to a device so that these variables and functions can be accessed or executed on the device.

Syntax

Syntax form 1

▶▶──#──pragma──omp declare target──▶◀

▶▶──*declaration-definition-seq*──▶◀
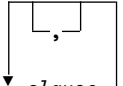
▶▶──#──pragma──omp end declare target──▶◀

Syntax form 2

▶▶──#──pragma──omp declare target──(──*extended-list*──)──────────────────────────▶◀

Syntax form 3

▶▶──#──pragma──omp declare target──┬─*clause*─┬───────────────────────────────────▶◀
                                   └──,──┘

where *clause* is **to**(*extended-list*). If *extended-list* is present with no clause, the **to** clause is implied.

extended-list in **Syntax form 2** and **Syntax form 3** is a comma-separated list of one or more extended list items. An extended list item is a variable, array section, or function name.

Usage

Declarations for global variables and functions in an **omp declare target** directive create device versions of the variables and functions and allocate storage on the device environment. Device variables can be accessed from target regions either directly or through an **omp declare target** function. The **to** clause maps the items when the device is initialized.

Restrictions
- The **omp declare target** directives must not be nested.
- You cannot specify the same item more than once.

- A variable type must be one of the allowed types in the **map** clause on target regions. For more information, see the *OpenMP Application Program Interface Language Specification*, which is available at http://www.openmp.org.
- All declarations and definitions for a function must have an **omp declare target** directive.
- `C++` You can specify overloaded functions or template functions only through an implicit extended list. The syntax of the **omp declare target** directive with an implicit extended list is **Syntax form 1.** `C++`
- The **omp declare target** and **omp threadprivate** directives are mutually exclusive.

Example

The following example shows how to specify variables and functions in an **omp declare target** directive.

```
#include <stdio.h>
#pragma omp declare target
struct vector {
  vector(int x, int y) : _x(x), _y(y) {}
  int dot(vector o) { return _x * o._x + _y*o._y; }
  int _x, _y;
} v1(1,1);
#pragma omp end declare target
int main() {
  vector v2(1,3);
  int res;
  #pragma omp target map(from:res)
  {   res = v1.dot(v2); }
  printf("(%d, %d) . (%d, %d) = %d\n", v1._x, v1._y, v2._x, v2._y, res);
}
```

The constructor function `vector::vector(int,int)`, nonstatic member function `vector::dot(vector)`, and static variable v1 are declared in the **omp declare target** directive. Thus, a device copy of each variable and function is created based on the host version. On entry to the target region, the variable v2 is implicitly mapped to and from the device, while the device copy of the variable v1 is accessed directly. The device version of the function `vector::dot` is invoked and the result is stored in the mapped variable `res`.

#pragma omp distribute
Purpose

The **omp distribute** directive specifies that the iterations of one or more loops will be executed by the thread teams in the context of their implicit tasks. The iterations are distributed across the master threads of all teams that execute the teams region to which the distribute region binds.

Syntax

```
>>--#--pragma--omp distribute------------------------------><
                        └─clause─┘
```

```
►►──block─────────────────────────────────────────────────────────────────────────►◄
```
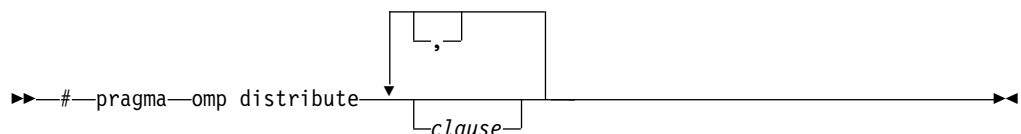
Parameters

clause is any of the following clauses:

collapse(*n***)**
> Specifies the number of loops that the **omp distribute** directive applies to. The expression that is represented by *n* must evaluate to a positive integer value. If no **collapse** clause is specified, the **omp distribute** directive applies only to the immediately following loop.

dist_schedule(static[,*chunk_size***])**
> - If **dist_schedule** is specified:
>   - If *chunk_size* is specified, groups of *chunk_size* iterations are assigned in a round-robin fashion to each participating team.
>   - If no *chunk_size* is specified, the iteration space is divided into approximately equal-sized chunks, and each chunk is assigned to each team. At most one chunk is assigned to each team.
> - If **dist_schedule** is not specified, iterations are distributed as if **dist_schedule(static)** is specified.

firstprivate(*list***)**
> Similar to **private**, except that the private copy is initialized with the value of the original variable. Data variables in *list* are separated by commas.

lastprivate(*list***)**
> Similar to **private**. In addition, the value of each new list item from the sequentially last iteration of the distribute loop is assigned back to the original list item. Data variables in *list* are separated by commas.

private(*list***)**
> Declares one or more list items to be private to the team in the enclosing team region. For every list item, a local copy is created for each team as if the variable was automatically declared with no initializers. All references to the original list item in the teams region are replaced with references to the private copy. A copy is created per team; hence, the copy is shared among the threads in the team. Data variables in *list* are separated by commas.

Usage

- The **omp distribute** directive takes effect only if you specify both the **-qsmp** and **-qoffload** compiler options.
- You must use the **omp distribute** directive on loops that are strictly nested within a teams region.
- There is no implicit barrier at the end of a **omp distribute** directive.

Example

```
#include <stdlib.h>
#include <omp.h>
int main()
{
const int N = 8;
int a[N];
int i;

#pragma omp target map(to: N) map(tofrom: a)
#pragma omp teams num_teams(2) thread_limit(N/2)
```

```
#pragma omp distribute
for (i=0; i<N; i++)
{
  a[i] = omp_get_team_num();
}
#pragma omp end distribute
#pragma omp end teams
#pragma omp end target
}
```

This target region contains a teams region that consists of two teams. The iterations of the closely nested distribute loop are assigned to these two teams, with the master thread of either team executing N/2 iterations.

Related reference:

#pragma omp distribute parallel for
Purpose

The **omp distribute parallel for** directive executes a loop using multiple teams where each team typically consists of several threads. The loop iterations are distributed across the teams in chunks in round robin fashion.

Syntax

Parameters

The **omp distribute parallel for** construct is a composite construct. *clause* can be any of the clauses that are accepted by the **omp distribute** or **omp parallel for** directive except the **linear** and **ordered** clauses. The specified **clause** has identical meanings and restrictions applied as used in the **omp distribute** or **omp parallel for** directive.

Usage

The **omp distribute parallel for** directive takes effect only if you specify both the **-qsmp** and **-qoffload** compiler options.

Rules

If any specified clause except the **collapse** clause is applicable to both the **omp distribute** and **omp parallel for** directives, it is applied twice; the **collapse** clause is applied only once.

The iterations of the loops that are associated with the **omp distribute parallel for** directive are distributed across the teams that bind to the loop construct. Each

team is assigned a chunk of the loop iterations. The size of the chunks is determined according to the clauses that apply to the **omp distribute** directive. Each chunk forms a parallel loop, and the parallel loop is distributed across the threads that participate in the team region according to the clauses that apply to the **omp parallel for** directive.

Examples

```
const int N = 8;
int A[N], B[N], C[N];
int k = 4;
int nteams = 16;
int block_threads = N/nteams;
for(int i=0; i<N; ++i)
{
   A[i] = 0;
   B[i] = i;
   C[i] = 3*i;
}
#pragma omp target map(tofrom: A) map(to: B, C)
#pragma omp teams num_teams(nteams)
#pragma omp distribute parallel for dist_schedule(static, block_threads)
for(int i=0; i<N; ++i)
{
  A[i] = B[i] + k*C[i];
}
```

In the beginning, the arrays A, B, and C and the scalar variables k, nteams, and block_threads are declared and initialized in the host environment.

Then, a target region is declared, and the arrays A, B, and C are explicitly mapped into the device environment. At the start of the target region, storage for A, B, and C is allocated on the device. The device copy of each array is then initialized with the content of the corresponding array on the host. The scalar variables k, nteams, and block_threads are implicitly mapped by the compiler as **firstprivate** because they are not explicitly mapped and the **defaultmap(tofrom:scalar)** clause is not present.

The target region is executed by the nteams teams of threads.

The loop iterations are first distributed across the teams in chunks of size equal to the value of the block_threads variable. Each chunk of iterations is further distributed across the threads in each team.

At the end of the target region, the copy of array A on the device is copied back into the host environment.

Related reference:

"-qsmp" on page 201

"-qoffload" on page 177

#pragma omp distribute parallel for simd
Purpose

The **omp distribute parallel for simd** directive distributes loop iterations to each master thread, further redistributes those iterations among the threads of each team, and then applies SIMD vectorization to each iteration.

Syntax

```
►►──#──pragma──omp distribute parallel for simd──┬──────────┬──►◄
                                                  │    ┌─,─┐ │
                                                  └─clause──┘

►►──for-loops────────────────────────────────────────────────────►◄
```
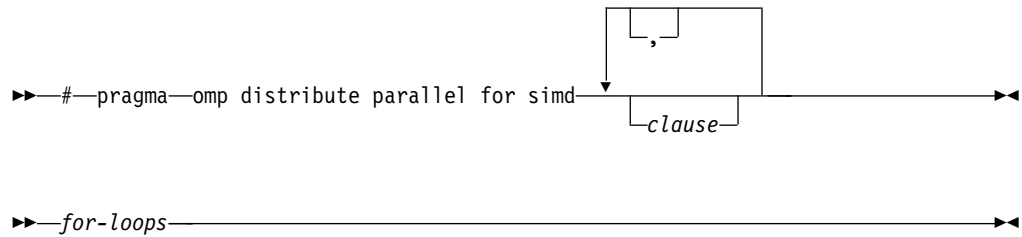
Parameters

The **omp distribute parallel for simd** construct is a composite construct. *clause* can be any of the clauses that are accepted by the **omp distribute** or **omp parallel for simd** directive with identical meanings and restrictions.

Usage

The **omp distribute parallel for simd** directive takes effect only if you specify both the **-qsmp** and **-qoffload** compiler options.

Rules

If any specified clause except the **collapse** clause is applicable to both the **omp distribute** and **omp parallel for simd** directives, it is applied twice; the **collapse** clause is applied only once.

You can specify only loop iteration variables on the **linear** clause.

Examples

```
int N = 8;
int a[N];
#pragma omp target map(to: N) map(tofrom: a)
#pragma omp teams num_teams(2) thread_limit(N/2)
#pragma omp distribute parallel for simd
for (i=0, i<N, i++)
{
  a[i] = N;
}
#pragma omp end distribute parallel for simd
#pragma omp end teams
#pragma omp end target
```

In this example, the target region contains a teams region that consists of two teams. With the **omp distribute parallel for simd** directive, the iterations of the closely nested distribute loop are assigned to the teams that are actually created, and the master thread of each team executes the distributed iterations. Each parallel-do SIMD chunk is further redistributed among threads of each team, and SIMD vectorization is applied to each iteration of the SIMD chunks.

Related reference:

"-qsmp" on page 201

"-qoffload" on page 177

#pragma omp distribute simd
Purpose

The **omp distribute simd** directive distributes loop iterations to each master thread and then executes each set of distributed iterations concurrently by using SIMD instructions.

Syntax

```
►►─#─pragma─omp distribute simd─┬───────────┬─►◄
                                └─clauses─┘
```

```
►►─for-loops─────────────────────────────────────────►◄
```
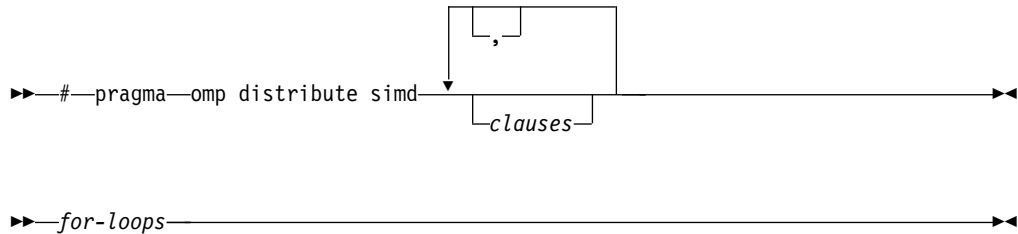
Parameters

The **omp distribute simd** construct is a composite construct. *clause* can be any of the clauses that are accepted by the **omp distribute** or **omp simd** directive with identical meanings and restrictions.

Usage

The **omp distribute simd** directive takes effect only if you specify both the **-qsmp** and **-qoffload** compiler options.

Rules

If any specified clause except the **collapse** clause is applicable to both the **omp distribute** and **omp simd** directives, the clause is applied twice; the **collapse** clause is applied only once.

You can specify only loop iteration variables on the **linear** clause.

Examples

```
const int N = 8;
int a[N];
int i;
#pragma omp target map(to: N) map(tofrom: a)
#pragma omp teams num_teams(2) thread_limit(N/2)
#pragma omp distribute simd
for (i=0; i<N; i++)
{
  a[i] = N;
}
#pragma omp end distribute simd
#pragma omp end teams
#pragma omp end target
```

In this example, the target region contains a teams region that consists of two teams. The iterations of the closely nested distribute loop are assigned to the teams that are actually created. With the **omp distribute simd** directive, the master thread of each team executes each set of distributed iterations concurrently by using SIMD instructions.

Related reference:
"-qsmp" on page 201
"-qoffload" on page 177

#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
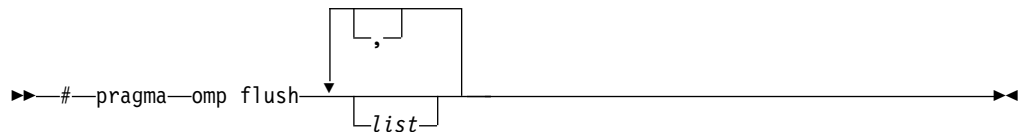
where *list* is a comma-separated list of variables that will be synchronized.

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 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

```
►►──#──pragma──omp for──┬─────────────┬──for-loops──────────────────►◄
                         │    ┌─ , ◄─┐ │
                         └──┴─clause─┴─┘
```
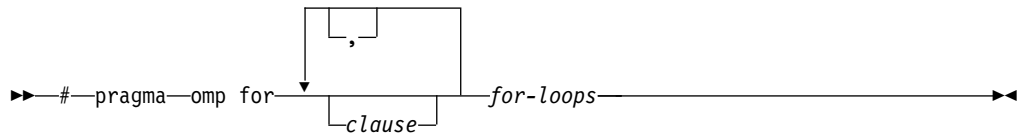
Parameters

clause is any of the following clauses:

collapse(*n*)
Allows you to parallelize multiple loops in a nest without introducing nested parallelism.

```
►►──COLLAPSE──(──n──)───────────────────────────────────►◄
```

- 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(*n***)**

Specify this clause if an ordered construct is present within the dynamic extent of the **omp for** directive.

```
▶▶──ORDERED─────────────────────────────────────────▶◀
              └─(─n─)─┘
```
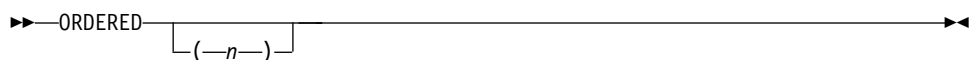
n is a constant positive integer representing the number of the loops that are associated with the loop construct. *n* does not affect how the logical iteration space is divided. If *n* is specified for the loop construct, the associated loops form a doacross loop nest, which is a loop nest that has cross-iteration dependence.

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.

Restriction:
- 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 for simd
Purpose

The **omp for simd** directive distributes the iterations of one or more associated loops across the threads that already exist in the team and indicates that the iterations executed by each thread can be executed concurrently using SIMD instructions.

Syntax

```
►►──#──pragma──omp for simd─┬────────────┬──────────────────────────►◄
                            │     ┌─,◄─┐  │
                            └──────clause─┘

►►──for-loops───────────────────────────────────────────────────────►◄
```

Parameters

The **omp for simd** construct is a composite construct. *clause* can be any of the clauses that are accepted by the **omp for** or **omp simd** directive. The specified `clause` has identical meanings and restrictions applied as used in the **omp for** or **omp simd** directive.

Rules

If any specified clause except the **collapse** clause is applicable to both the **omp for** and **omp simd** directives, it is applied twice; the **collapse** clause is applied only once.

A list item can only appear in a **linear** or **firstprivate** clause, but not both.

The iterations of the loops that are associated with the **omp for simd** directive are distributed across the teams that bind to the loop construct, with any clauses that apply to the **omp for** directive in effect. The resulting chunks of iterations then are converted to a SIMD loop, with any clauses that apply to the **omp simd** directive in effect.

Usage

You can take advantage of the **omp for simd** directive by nesting it inside the **omp parallel** construct or using the combined construct **omp parallel for simd**.

Examples

```
int N = 8;
int a[N];
#pragma omp target map(to: N) map(tofrom: a)
#pragma omp parallel for simd
for (i=0, i<N, i++)
{
 a[i] = N;
}
#pragma omp end parallel for simd
#pragma omp end target
```

In this example, the **omp parallel for simd** directive distributes the iterations of the loop across the threads and executes the iterations in parallel. Then, the iterations are allowed to be executed by each thread concurrently by using SIMD instructions.

#pragma omp master
Purpose

The **omp master** directive identifies a section of code that must be run only by the master thread.

Syntax

▶▶──#──pragma──omp master──▶◀

Usage

Threads other than the master thread will not execute the statement block associated with this construct.

No implied barrier exists on either entry to or exit from the master section.

#pragma omp ordered
Purpose

The **omp ordered** directive identifies a structured block of code that must be executed in sequential order.
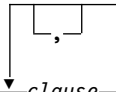
Syntax

Syntax form 1

▶▶──#──pragma──omp ordered───▶◀

Syntax form 2

▶▶──#──pragma──omp ordered──┬─*clause*─┬──────────────────────────────────▶◀
　　　　　　　　　　　　　　　　└──,──┘

where *clause* is **depend**. **depend** clauses specify the order in which threads in a team execute **ordered** regions. The **depend** clause takes one of the following forms:

depend(source)
> **source** specifies the satisfaction of cross-iteration dependences that arise from the current iteration.

depend(sink : *vec*)
> **sink** specifies a cross-iteration dependence, where the iteration vector *vec* indicates the iteration that satisfies the dependence. *vec* has the following form:
>
> $x_1 [\pm d_1], x_2 [\pm d_2], \dots , x_n [\pm d_n]$

where *n* is the value specified by the **ordered** clause in the loop directive, x_i denotes the loop iteration variable of the *i*-th nested loop associated with the loop directive, and d_i is a non-negative constant integer.

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
Purpose

The **omp parallel** directive explicitly instructs the compiler to parallelize the chosen block of code.

Syntax

```
►►─#─pragma─omp parallel──┬──────────┬──────────────────────────────►◄
                          │    ,     │
                          └─clause───┘
```

Parameters

clause is any of the following 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.

`proc_bind(master | close | spread)`

Specifies a policy for assigning threads to places within the current place partition. At most one `proc_bind` clause can be specified on the parallel directive. If the **OMP_PROC_BIND** environment variable is not set to **FALSE**, the proc_bind clause overrides the first element in the **OMP_PROC_BIND** environment variable. If the **OMP_PROC_BIND** environment variable is set to **FALSE**, the proc_bind clause has no effect.

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.

By default, nested parallel regions are serialized.

Related information:

"OMP_NESTED" on page 28

"OMP_PROC_BIND" on page 32

#pragma omp section, #pragma omp sections
Purpose

The **omp sections** directive distributes work among threads bound to a defined parallel region.

Syntax

```
>>--#--pragma--omp sections--+--clause--+-------------------------><
                             '--,-------'
```

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 simd
Purpose

The **omp simd** directive is applied to a loop to indicate that multiple iterations of the loop can be executed concurrently by using SIMD instructions.

Syntax

```
         ┌──── , ◄────┐
►►──#──pragma──omp simd──┴──┬────────┬──┴──────────────────────►◄
                            └─clause─┘


►►──for-loops──────────────────────────────────────────────────►◄
```
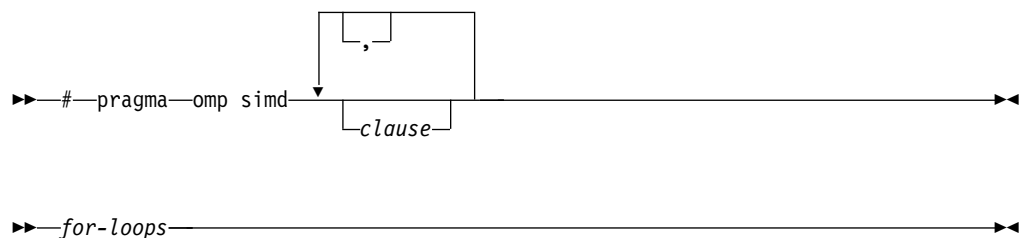
Parameters

clause is any of the following clauses:

aligned(*list*[:*alignment*])
  Declares that the object to which argument in *list* points is byte aligned
  according to the number of bytes expressed by *alignment*. *alignment* must be a
  constant positive integer expression. If *alignment* is not specified, the alignment
  is defined by the target platforms. A list item cannot appear in more than one
  aligned clause.

collapse(*n*)[1]
  Specifies the number of loops that the **omp simd** directive applies to. The
  expression that is represented by *n* must be a constant positive integer
  expression. If the collapse clause is not specified, the **omp simd** directive
  applies to only the loop that immediately follows.

lastprivate(*list*)
  Declares the data variables in *list* to be private to each SIMD lane. The final
  value of each variable in *list* is as follows:

  • If assigned a value, each variable has the value that is assigned to that
    variable in the sequentially last iteration.

  • If not assigned a value, each variable has an indeterminate value.

  Data variables in *list* are separated by commas.

linear(*list*[:*linear-step*])
  Declares the data variables in *list* to be private to each SIMD lane and to have
  a linear relationship with the iteration space [2] of a loop. The value of the new
  list item on each iteration of the associated loops is the result of adding the
  following values:

  • The value of the original list item before entering the construct

  • The product of the logical number of the iteration and *linear-step*

  The final value of each variable in *list* has the value that is assigned to that
  variable in the sequentially last iteration. The default value for *linear-step* is 1.

private(*list*)
  Declares the data variables in *list* to be private to each SIMD lane. Data
  variables in *list* are separated by commas.

reduction(*reduction-identifier*:*list*)
  Performs a reduction on each data variable in *list* according to
  reduction-identifier.

  The clause creates a private copy for data variables in *list* for each SIMD lane,
  initializes the private copies with the initializer value of *reduction-identifier*, and
  updates the original list item after the end of the region with the values of the
  private copies by using the combiner that is associated with *reduction-identifier*.

  For more information, see the *OpenMP Application Program Interface Language
  Specification*, which is available at http://www.openmp.org.

safelen(*length*)[1] [3]
  Indicates that no two iterations that are executed concurrently by using SIMD
  instructions can have a distance in the logical iteration space [2] greater than the
  value expressed by *length*. *length* must be a constant positive integer
  expression.

simdlen(*length*)[1] [3]
  Specifies the preferred number of iterations to be executed concurrently. The

number of SIMD loop iterations that are executed concurrently is implementation defined. Each concurrent iteration is executed by a different SIMD lane. You must preserve lexical forward dependencies in the iterations of the original loop within each set of concurrent iterations. *length* must be a constant positive integer expression.

Notes:

1. You can specify this clause only once for each **omp simd** directive.
2. Iterations in a SIMD loop are logically numbered from 0 to N-1, where N is the number of loop iterations. The logical numbering denotes the sequence in which the iterations would be executed if the associated loop was executed not using SIMD instructions.
3. If you specify both the **simdlen** and **safelen** clauses, the value of the **simdlen** parameter must be less than or equal to the value of the **safelen** parameter.

Rules

All loops that are associated with the construct must be perfectly nested; that is, you cannot insert any intervening code or OpenMP directive between any two loops.

The associated loops must be structured blocks.

A program that branches into or out of a simd region is nonconforming.

An ordered construct with the simd clause is the only OpenMP construct that can be encountered during the execution of a simd region.

Limitation: The **omp declare simd** directive is not supported in this release.

Examples

Example 1

The following SIMD construct enables the execution of multiple iterations of the associated loop concurrently by using SIMD instructions.

```
#pragma omp simd
{
  for (i=0; i<N; i++) {
    a[i] = a[i] + b[i] * c[i];
  }
}
```

Example 2

The loop in the following example contains a lexically forward loop-carried dependency that prohibits concurrent execution of all iterations of the loop. The omp simd safelen(4) directive specifies that the loop iterations that are at a distance of four or less in the logical iteration space can be executed in parallel by using SIMD instructions.

```
#pragma omp simd safelen(4)
{
  for (i=0; i<(N-4); i++) {
    a[i] = a[i+4] + b[i] * c[i];
  }
}
```

#pragma omp single
Purpose

The **omp single** directive identifies a section of code that must be run by a single available thread.

Syntax
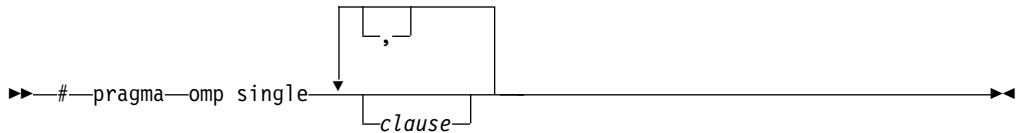
Parameters

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 target
Purpose

The **omp target** directive instructs the compiler to generate a *target task*, that is, to map variables to a device data environment and to execute the enclosed block of code on that device.

Use the **omp target** directive to define a *target region*, which is a block of computation that operates within a distinct data environment and is intended to be offloaded onto a parallel computation device during execution.

Syntax

```
               ┌─────────┐
               │   ┌─,─┐ │
               ▼   └───┘ │
►►──#──pragma──omp target─┬───────┬──────────────────────────────►◄
                          └─clause─┘
```

```
►►──block───────────────────────────────────────────────────────►◄
```

Parameters

clause is any of the following clauses:

defaultmap(tofrom:scalar)
> Changes the default implicit mapping rule from **firstprivate** to **tofrom** for scalar variables. For more information about implicit mapping, see the Rules section.

depend(_dependence-type_**:**_list_**)**
> Establishes scheduling dependences between the target task and sibling tasks that share list items. The *dependence-type* can be **in**, **out**, or **inout**. Data variables in *list* are separated by commas.
> - If *dependence-type* is **in** or **inout**, for each list item that has the same storage location as a list item in a depend clause of a sibling task with the **out** or **inout** dependence type, a scheduling dependence for the target task on the sibling task is created.
> - If *dependence-type* is **out** or **inout**, for each list item that has the same storage location as a list item in a depend clause of a sibling task with the **in**, **out**, or **inout** dependence type, a scheduling dependence for the target task on the sibling task is created.

device(_exp_**)**
> Creates the data environment on the device of ID *exp*. *exp* is an integer expression that evaluates to a non-negative integer value less than the value of `omp_get_num_devices()`.

firstprivate(_list_**)**
> Declares the data variables in *list* to be private to the target task and shared by every thread team that runs the region. A new item is created for each list item

that is referenced by the target task. Each new data variable is initialized with the value of the original variable at the time the target construct is encountered. Data variables in *list* are separated by commas.

if([target:]*exp***)**
When the **if** clause is specified and the scalar expression that is represented by *exp* evaluates to zero, the target region is executed by the host device in the host data environment.

is_device_ptr(*list***)**
Indicates that the data variables in *list* are device pointers that exist within the device data environment. Supported types are pointers, references to pointers, arrays, and references to arrays.

map([[*map-type-modifier***[,]]***map-type:]***list***)**
Specifies the data variables in *list* to be explicitly mapped from the original variables in the host data environment to the corresponding variables in the device data environment of the device specified by the construct.

The *map-type* can be **to**, **from**, **tofrom**, or **alloc**.

- If a list item does not exist in the device data environment, a new item is created in the device data environment.
  - If *map-type* is **to** or **tofrom**, this new item is initialized with the value of the original list item in *list* in the host data environment.
  - If *map-type* is **from** or **alloc**, the initial value of the list item in the device data environment is undefined.

  On exit from the target region, if storage for the list item was created in the device data environment when this construct was first encountered, the list item is deallocated from the device data environment. Furthermore, if *map-type* is **from** or **tofrom**, the original list item is updated with the current value of the corresponding list item in the device data environment before the list item in the device data environment is deallocated.

- If the list item exists in the device data environment when the construct is encountered, the allocation count of the item in the device environment changes as follows:
  - It is incremented by one at the start of the construct
  - It is decremented by one at the end of the construct.

The *map-type-modifier* is **always**. If this modifier is specified, the following rules apply:

- If *map-type* is **to** or **tofrom**, the value of the original list item is always copied to the device environment, regardless of whether a new item was created in the device data environment for the list item.
- If *map-type* is **from** or **tofrom**, the value of the list item is always copied from the device environment to the original list item, regardless of whether the device list item will be deallocated at termination of the construct.

nowait
Eliminates the implicit barrier so the parent task can make progress even if the target task is not yet completed. By default, an implicit barrier exists at the end of the target construct, which ensures the parent task cannot continue until the target task is completed.

private(*list***)**
Declares the data variables in *list* to be private to the target task and shared by

every thread team that runs the region. A new item is created for each list item that is referenced by the target task. Data variables in *list* are separated by commas.

reduction(*reduction-identifier***:***list***)**
Specifies that for each data variable in *list*, a private copy is created and initialized based on the *reduction-identifier*. At the end of the region, the original data variable is updated with the values of the private copies using a combiner based on the *reduction-identifier*.

> **Limitations:**
> - The reduction clause on the **omp target** directive is supported only in the form of combined constructs, such as **omp target parallel for** or **omp target teams distribute parallel for**.
> - Data variables in *list* cannot be arrays or array sections.
> - User-defined reductions are not supported.

Usage

To enable the **omp target** directive to execute the target region, you must specify the **-qsmp** and **-qoffload** options to offload the region to the device environment. If a target region cannot be successfully offloaded to a device, the target region is executed within the host environment.

Rules

Nesting of target regions, either dynamically or statically, is not allowed.

General mapping rules are as follows:
- Pointers are mapped as zero-length array sections with zero base for both explicit and implicit mapping.
- If a zero-length array section that is derived from a pointer variable is mapped, that variable is initialized with the address of the corresponding storage location on the device. If the corresponding storage does not exist, that is, it has not been mapped before, the pointer variable is initialized to NULL.

The data environment of a target region is defined by the implicit and explicit mapping of variables between the host and device:

Implicit mapping
The compiler determines which variables must be mapped to, from, or both to and from the device data environment. Scalar variables that are not explicitly mapped are implicitly mapped as **firstprivate** if **defaultmap(tofrom:scalar)** is not specified.

Explicit mapping
You can use the **map** clause on the target region to explicitly list variables to be mapped to, from, or both to and from the device data environment.

A listed data variable cannot appear in both a data-sharing clause and the map clause on the same target construct.

Examples

```
int main()
{
  int x = 1;
  #pragma omp target map(toform: x)
```

```
  x = x + 1; // The copy of x on the device has a value of 2.
  printf("After the target region is executed, x = %d\n", x);
  return 0;
}
```

The integer x is declared in the host environment, and its initial value is set to 1 on the host. The target region is declared with explicit map type **tofrom** of x, so the storage for x is allocated on the device and the device copy of x is initialized to 1. Within the target region, the value of the copy of x on the device is incremented by 1. At the end of the target region, x is mapped back to the host environment according to the map type **tofrom**, and the host prints the value of x to be 2.

Related reference:

"-qoffload" on page 177

"-qsmp" on page 201

"omp_target_alloc" on page 563

#pragma omp target data
Purpose

The **omp target data** directive maps variables to a device data environment, and defines the lexical scope of the data environment that is created. The **omp target data** directive can reduce data copies to and from the offloading device when multiple target regions are using the same data.

Syntax

```
                                   ┌─ , ─┐
                                   │     │  (1)
►►──#──pragma──omp target data──▼─clause─┴─────────────────────────►◄
```
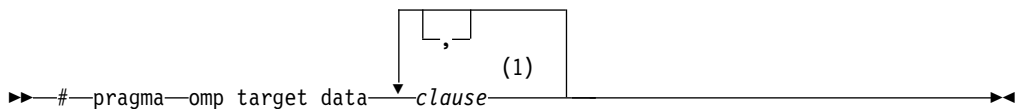
Notes:

1   You must specify at least one **map** or **use_device_ptr** clause.

```
►►──block───────────────────────────────────────────────────────►◄
```

Parameters

clause is any of the following clauses:

if([target data:] *scalar-expression***)**
    When an **if** clause is present and the **if** clause expression evaluates to zero, the device is the host.

    At most one **if** clause can appear on the directive.

device(*integer-expression***)**
    Creates the data environment on the device with the designated ID. The *integer-expression* must evaluate to a non-negative integer value less than the value of omp_get_num_devices().

map([[*map-type-modifier***[,]]***map-type:***]***list***)**
    Specifies the data variables in *list* to be explicitly mapped from the original variables in the host data environment to the corresponding variables in the device data environment of the device specified by the construct.

The *map-type* can be **to**, **from**, **tofrom**, or **alloc**.

- If a list item does not exist in the device data environment, a new item is created in the device data environment.
  - If *map-type* is **to** or **tofrom**, this new item is initialized with the value of the original list item in *list* in the host data environment.
  - If *map-type* is **from** or **alloc**, the initial value of the list item in the device data environment is undefined.

  On exit from the target data region, if storage for the list item was created in the device data environment when this construct was first encountered, the list item will be deallocated from the device data environment. Furthermore, if the *map-type* is **from** or **tofrom**, the original list item is updated with the current value of the corresponding list item from the device data environment before deallocating the list item from the device data environment.

- If the list item was already present in the device data environment when the construct is encountered, the allocation count of the item in the device environment changes as follows:
  - It is incremented by one at the start of the construct
  - It is decremented by one at the end of the construct.

The *map-type-modifier* is **always**. If this modifier is present, the following rules apply:

- If *map-type* is **to** or **tofrom**, the value of the original list item is always copied to the device environment, regardless of whether a new item was created in the device data environment for the list item.

- If *map-type* is **from** or **tofrom**, the value of the list item is always copied from the device environment to the original list item, regardless of whether the device list item will be deallocated at the termination of the construct.

At most one **map** clause can appear on the directive.

use_device_ptr(*list*)
Converts the data variables in *list* into device pointers to the corresponding variables in the device data environment. References to variables in *list* throughout the target data region must only be to the address of those variables. You can use only direct access variables; object members and indirectly accessed variables are not supported by the **use_device_ptr** clause.

Usage

The **omp target data** directive takes effect only if you specify both the **-qsmp** and **-qoffload** compiler options.

Rules

The target data construct creates a device data environment, for the duration of the execution of its lexical scope by the encountering thread, during which the storage, and values, of the mapped list items are present on the device.

Example

```
double *array = (double*)malloc(sizeof(double)*N);
// Target data region
#pragma omp target data map(from: array[0:N])
{
  // The first target region
  #pragma omp target map(tofrom: array[0:N])
```

```
   {
      for (int i=0; i<N; i++)array[i] = double(i) / N;
   }
   // The second target region
   #pragma omp target map(tofrom: array[0:N])
   {
      for (int i=0; i<N; i++) array[i] = 1.0 / array[i];
   }
}
for (int i=0; i<N; i++) sum += array[i];
```

At first, storage for an array is allocated in the host environment, but this storage is not initialized. When the target data region is encountered, corresponding storage for the array is created on the device. The device storage is also not initialized.

When the first target region is encountered, the OpenMP runtime will check whether the storage corresponding to the array already exists on the device. No further action, with respect to the device storage, is taken because the corresponding storage already exists on the device when the target region is encountered. The device storage of the array is then initialized, and the target region completes. Upon completion, the OpenMP runtime will recognize that the storage of the array should remain on the device, so no copy-back from device to host will occur. At this point, the array storage on the host remains uninitialized, while the array storage on the device is initialized.

The same happens when the second target region is encountered. No copies between host and device will occur because the storage is already present on the device when the target construct is encountered, and because that storage is to remain on the device after completion of the target region. After execution of this target region is completed, the array storage on the host remains uninitialized and the array storage on the device has been altered.

Finally, the end of the target data lexical scope is encountered. This causes the values of the array storage on the device to be copied back to the host, and the device storage to be deallocated. After the lexical scope of the target data region is executed, only the host storage of the array exists, and it contains the values that were calculated on the device.

In contrast, see the following example that doesn't contain a target data construct.
```
double *array = (double*)malloc(sizeof(double)*N);
#pragma omp target map(tofrom: array[0:N])
{
   for (int i=0; i<N; i++)array[i] = double(i) / N;
}
#pragma omp target map(tofrom: array[0:N])
{
   for (int i=0; i<N; i++) array[i] = 1.0 / array[i];
}
for (int i=0; i<N; i++) sum += array[i];
```

When each of the two target regions is encountered, the OpenMP runtime takes the following actions:

1. Allocate storage on the device corresponding to the array.
2. Copy the values of the array from the host to the device.
3. Execute the region on the device.
4. Copy the values of the array from the device to the host.
5. Deallocate the device storage of the array.

Related reference:

## #pragma omp target enter data
Purpose

The **omp target enter data** directive maps variables to a device data environment. The **omp target enter data** directive can reduce data copies to and from the offloading device when multiple target regions are using the same data, and when the lexical scope requirement of the **omp target data** construct is not appropriate for the application.

Syntax
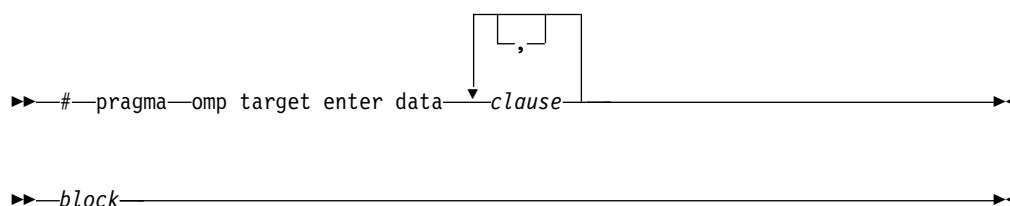
Note: You must specify at least one *map* clause.

Parameters

clause is any of the following clauses:

if([**target enter data:**] *scalar-expression***)**
When an **if** clause is present and the **if** clause expression evaluates to zero, the device is the host.

At most one **if** clause can appear on the directive.

depend(*dependence-type***:***list***)**
Establishes scheduling dependences between the target task and sibling tasks that share list items. The *dependence-type* can be **in**, **out**, or **inout**. Data variables in *list* are separated by commas.
- If *dependence-type* is **in** or **inout**, for each list item that has the same storage location as a list item in a depend clause of a sibling task with the **out** or **inout** dependence type, a scheduling dependence for the generated task on the sibling task is created.
- If *dependence-type* is **out** or **inout**, for each list item that has the same storage location as a list item in a depend clause of a sibling task with the **in**, **out**, or **inout** dependence type, a scheduling dependence for the generated task on the sibling task is created.

device(*integer-expression***)**
Creates the data environment on the device with the designated ID. The *integer-expression* must evaluate to a non-negative integer value that is less than the value of omp_get_num_devices().

map([[*map-type-modifier***[,]**]*map-type***:**]*list***)**
Specifies the data variables in *list* to be explicitly mapped from the original variables in the host data environment to the corresponding variables in the device data environment of the device specified by the construct.

The *map-type* can be **to** or **alloc**.

- If a list item does not exist in the device data environment, a new item is created in the device data environment.
  - If *map-type* is **to**, this new item is initialized with the value of the original list item in *list* in the host data environment.
  - If *map-type* is **alloc**, the initial value of the list item in the device data environment is undefined.
- If the list item was already present in the device data environment when the construct is encountered, the allocation count of the item in the device environment is decremented by one.

The *map-type-modifier* is **always**. If this modifier is present, the value of the original list item is always copied to the device environment if the *map-type* is **to**, regardless of whether a new item was created in the device data environment for the list item.

nowait
> Enables the target enter data construct to perform asynchronously with respect to the encountering thread. By default, the encountering thread must wait for the completion of the construct.

Usage

The **omp target enter data** directive takes effect only if you specify both the **-qsmp** and **-qoffload** compiler options.

Rules

The target enter data construct adds mapped list items to the device data environment.

Example

```
double *array = (double*)malloc(sizeof(double)*N);
#pragma omp target enter data map(alloc: array[0:N])
#pragma omp target map(tofrom: array[0:N])
{
  #pragma omp target map(tofrom: array[0:N])
  {
    for (int i=0; i<N; i++)array[i] = double(i) / N;
  }
  #pragma omp target map(tofrom: array[0:N])
  {
    for (int i=0; i<N; i++) array[i] = 1.0 / array[i];
  }
}
#pragma omp target update from(array[0:N])
for (int i=0; i<N; i++) sum += array[i];
```

At first, storage for an array is allocated in the host environment, but this storage is not initialized. When the target enter data construct is encountered, corresponding storage for the array on the device is created. This device storage is also not initialized.

When the first nested target region is encountered, the OpenMP runtime will check whether storage corresponding to the array already exists on the device. No further action, with respect to the device storage, is taken because the corresponding storage already exists on the device when the target region is encountered. The device storage of the array is then initialized, and the target

region completes. Upon completion, the OpenMP runtime will recognize that the storage of the array should remain on the device, and so no copy-back from device to host will occur. At this point, the array storage on the host remains uninitialized, and the array storage on the device is initialized.

The same happens when the second nested target region is encountered. No copies between host and device will occur because the storage is already present on the device when the target construct is encountered, and because that storage is to remain on the device after completion of the target region. After execution of this target region is completed, the array storage on the host remains uninitialized and the array storage on the device has been altered.

Finally, the target update construct is encountered. This causes the values of the array storage on the device to be copied back to the host. Now, both host and device have the storage for the array, and both of these copies contain the same values.

Related reference:

"-qoffload" on page 177

"-qsmp" on page 201

#pragma omp target exit data
Purpose

The **omp target exit data** directive unmaps variables from a device data environment. The **omp target exit data** directive can limit the amount of device memory when you use the **omp target enter data** construct to map items to the device data environment.

Syntax
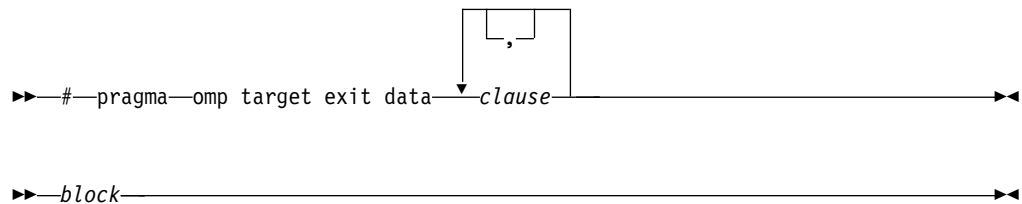
```
            ┌─── , ◄──┐
            │         │
►►─#─pragma─omp target exit data──▼─clause─┴──────────────────►◄


►►─block──────────────────────────────────────────────────────►◄
```

Note: You must specify at least one *map* clause.

Parameters

clause is any of the following clauses:

if([target exit data:] *scalar-expression***)**
> When an **if** clause is present and the **if** clause expression evaluates to zero, the device is the host.
>
> At most one **if** clause can appear on the directive.

depend(*dependence-type:list***)**
> Establishes scheduling dependences between the target task and sibling tasks that share list items. The *dependence-type* can be **in**, **out**, or **inout**. Data variables in *list* are separated by commas.

- If *dependence-type* is **in** or **inout**, for each list item that has the same storage location as a list item in a depend clause of a sibling task with the **out** or **inout** dependence type, a scheduling dependence for the generated task on the sibling task is created.
- If *dependence-type* is **out** or **inout**, for each list item that has the same storage location as a list item in a depend clause of a sibling task with the **in**, **out**, or **inout** dependence type, a scheduling dependence for the generated task on the sibling task is created.

device(*integer-expression***)**
> Creates the data environment on the device with the designated ID. The *integer-expression* must evaluate to a non-negative integer value less than the value of `omp_get_num_devices()`.

map([[*map-type-modifier***[,]]***map-type:***]***list***)**
> Specifies the data variables in *list* to be explicitly mapped from the original variables in the host data environment to the corresponding variables in the device data environment of the device specified by the construct.
>
> The *map-type* can be **from**, **release**, or **delete**.
> - If a list item does not exist in the device data environment, then this construct does nothing with respect to that list item.
>   - If *map-type* is **from** or **release**, the allocation count of the item in the device environment is decremented by one.
>   - If *map-type* is **delete**, the item's allocation count in the device data environment is set to zero.
>
>   If this construct reduces an item's allocation count in the device data environment to zero, then the item is deallocated from the device data environment.
> - If this construct deallocates a list item from the device data environment, and the *map-type* is **from**, then the values for that list item are copied from the device data environment to the host prior to deallocation of the list item on the device.
>
> The *map-type-modifier* is **always**. If this modifier is present, the value of the original list item is always assigned the value of the corresponding list item on the device environment if the *map-type* is **from**, regardless of whether a new item was created in the device data environment for the list item.

nowait
> Enables the target exit data construct to perform asynchronously with respect to the encountering thread. By default, the encountering thread must wait for the completion of the construct.

Usage

The **omp target exit data** directive takes effect only if you specify both the **-qsmp** and **-qoffload** compiler options.

Rules

The target exit data construct decrements the allocation count of mapped list items in the device data environment. When the allocation count of an item reaches zero, it is deallocated from the device data environment and its value might be copied back to the host environment.

Example

```
double *array = (double*)malloc(sizeof(double)*N);
#pragma omp target enter data map(alloc: array[0:N])
#pragma omp target map(tofrom: array[0:N])
{
  #pragma omp target map(tofrom: array[0:N])
  {
   for (int i=0; i<N; i++)array[i] = double(i) / N;
  }
  #pragma omp target map(tofrom: array[0:N])
  {
   for (int i=0; i<N; i++) array[i] = 1.0 / array[i];
  }
}
#pragma omp target exit data map(from: array[0:N])
for (int i=0; i<N; i++) sum += array[i];
```

At first, storage for an array is allocated in the host environment, but this storage is not initialized. When the target enter data construct is encountered, corresponding storage for the array is created on the device. This device storage is also not initialized.

When the first nested target region is encountered, the OpenMP runtime will check whether the storage corresponding to the array already exists on the device. No further action, with respect to the device storage, is taken because the corresponding storage already exists on the device when the target region is encountered. The device storage of the array is then initialized, and the target region completes. Upon completion, the OpenMP runtime will recognize that the storage of the array should remain on the device, so no copy-back from device to host will occur. At this point, the array storage on the host remains uninitialized, and the array storage on the device is initialized.

The same happens when the second nested target region is encountered. No copies between host and device will occur because the storage is already present on the device when the target construct is encountered, and because that storage is to remain on the device after completion of the target region. After execution of this target region is completed, the array storage on the host remains uninitialized and the array storage on the device has been altered.

Finally, the target exit data construct is encountered. The array storage in the device data environment will be deallocated. This causes the values of the array storage on the device to be copied back to the host because the *map-type* is **from**.

Now, only the host has the storage for this array, and the values in the array are the same as those calculated by the offload device.

Related reference:

"-qoffload" on page 177
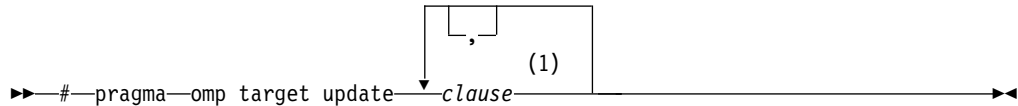
"-qsmp" on page 201

#pragma omp target update
Purpose

The **omp target update** directive makes the list items in the device data environment consistent with the original list items by copying data between the host and the device.

The direction of data copying is specified by *motion-type*.

Syntax

```
>>--#--pragma--omp target update----clause----------------------><
                                   ^      (1)
                                   |  ,  |
```

Notes:

1   You must specify at least one *motion-type* clause.

Parameters

clause is any of the following clauses:

motion-type(*list***)**
> Synchronizes the values of the list items between the host and device data environment. The *motion-type* can be **from** or **to**. A list item can only appear in a **from** or **to** clause, but not both. The copying occurs to each list item that has storage on the device data environment or each original list item on the host data environment according to the value of *motion-type*:
>
> * If *motion-type* is **from**, the value of each list item is copied from the corresponding item on the device to the original list item on the host.
> * If *motion-type* is **to**, the value of each list item is copied from the original list item on the host to the corresponding item on the device.
>
> If the corresponding list item is not present in the device data environment, no data movement happens.

depend(*dependence-type***:***list***)**
> Establishes scheduling dependences between the target task and sibling tasks that share list items. The *dependence-type* can be **in**, **out**, or **inout**. Data variables in *list* are separated by commas.
>
> * If *dependence-type* is **in** or **inout**, for each list item that has the same storage location as a list item in a depend clause of a sibling task with the **out** or **inout** dependence type, a scheduling dependence for the generated task on the sibling task is created.
> * If *dependence-type* is **out** or **inout**, for each list item that has the same storage location as a list item in a depend clause of a sibling task with the **in, out**, or **inout** dependence type, a scheduling dependence for the generated task on the sibling task is created.

device(*exp***)**
> Creates the data environment on the device of ID *exp*. The integer expression *exp* must evaluate to a non-negative integer value less than the value of `omp_get_num_devices()`. You can specify at most one **device** clause.

if([target:]*exp***)**
> When the **if** clause is specified and the scalar expression *exp* evaluates to zero, no copying of data occurs. You can specify at most one **if** clause.

nowait
> Enables the target update construct to execute asynchronously with respect to the encountering thread. By default, the encountering thread must wait for the completion of the construct.

Usage

The **omp target update** directive takes effect only if you specify both the **-qsmp** and **-qoffload** compiler options.

Examples

```
int main (){
    int x;
    x = 0;
    #pragma omp target data map(tofrom: x)
//  A device data environment is created, and x in the device data environment
//  is initialized. See status 1.
    {
       x = 10;
// See status 2.
       #pragma omp target update to(x)
// See status 3.
    }
}
```

You can find the variable values for each status in the following table.

Table 29. Variable values on the host and device

Status	Value of x on the host	Value of x on the device
Status 1	0	0
Status 2	10	0
Status 3	10	10

Related reference:

"-qoffload" on page 177

"-qsmp" on page 201

#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

```
>>--#--pragma--omp task--▼--clause--+----------------------------><
                         ^--,--+
```

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

depend(_dependence-type_**:**_list_**)**
    Establishes scheduling dependences between sibling tasks that share list items. The _dependence-type_ can be **in**, **out**, or **inout**. Data variables in _list_ are separated by commas.

- If _dependence-type_ is **in** or **inout**, for each list item that has the same storage location as a list item in a depend clause of a sibling task with the **out** or **inout** dependence type, a scheduling dependence for the generated task on the sibling task is created.
- If _dependence-type_ is **out** or **inout**, for each list item that has the same storage location as a list item in a depend clause of a sibling task with the **in**, **out**, or **inout** dependence type, a scheduling dependence for the generated task on the sibling task is created.

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 taskyield——————————————————————————————►◄
```

#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

```
►►──#──pragma──omp taskwait──────────────────────────────────────────►◄
```

Related reference:

#pragma omp teams
Purpose

The **omp teams** directive creates a collection of thread teams. The master thread of each team executes the teams region.

Syntax

```
                              ┌─────┐
                              │ ┌─┐ │
                              ▼ │,│ │
►►──#──pragma──omp teams──────┴─┴─┴─┴──────────────────────────────────►◄
                        └─clause─┘


►►──block───────────────────────────────────────────────────────────────►◄
```

Parameters

clause is any of the following clauses:

default(none│shared)
> Specifies the default attribute of variables referenced inside the teams region. Specifying **default(none)** requires variables with no implicit data sharing attributes to be specified on **firstprivate**, **private** or **shared** clause.

firstprivate(*list***)**
> Similar to the *private* clause, except the private copy is initialized with the value of the original variable. Data variables in *list* are separated by commas.

num_teams(*exp***)**
> Specifies an upper limit on the number of teams created. You can obtain the number of teams in a thread by calling the **omp_get_num_teams** function. The expression represented by *exp* must evaluate to a positive integer value.

private(*list***)**
> Declares one or more list items to be private to the team. For every list item, a local copy is created for each team as if the variable was an automatic declared with no initializers. All references to the original list item in the teams region are replaced with references to the private copy. A copy is created per team; hence that copy is shared among the threads in the team. Data variables in *list* are separated by commas.

reduction(*reduction-identifier***:** *list***)**
> Specifies a reduction operator and a list of reduction variables. For each reduction variable, a private copy is created and initialized with the value of the *reduction-identifier*. At the end of the teams region, the master threads of all

teams perform a reduction into the value of the original variable. Data variables in *list* are separated by commas.

shared(*list***)**

Declares one or more list items to be shared among the teams. Data variables in *list* are separated by commas.

thread_limit(*exp***)**

Specifies an upper limit on the number of threads inside each team. The expression represented by *exp* must evaluate to a positive integer value.

Usage

- For a target region without specifying *teams*, a single team is created and its master thread executes the target region.
- The **omp teams** directive must be strictly nested inside the target region, that is, no code can exist between the **omp target** directive and the **omp teams** directive. You must strictly nest the following OpenMP regions inside the teams region:
  - **omp distribute**
  - **omp distribute simd**
  - **omp distribute parallel for**
  - **omp distribute parallel for simd**
  - Parallel regions, including any parallel regions arising from combined constructs
- Basically, the compiler creates as many teams as you request. However, if the number you request exceeds either of the following numbers, teams of the smaller number between the two are created:
  - The value specified on the **num_teams** clause
  - 65536, which is the implementation defined limit
- Basically, the compiler creates as many threads per team as you request. However, if the number you request exceeds either of the following numbers, threads of the smaller number between the two are created per team:
  - The value specified on the **thread_limit** clause
  - 992, which is the implementation defined limit
- After the teams are created, the number of teams remains constant for the duration of the teams region.
- When the **omp teams** directive is used as part of a combined construct, for example, the **omp target teams** directive, the expression used in the **num_teams** or **thread_limit** clause is evaluated on the host.
- Within a teams region, each team has a unique team number. Team numbers are consecutive integers that range from zero to one less than the number of teams. You can obtain the team number by calling the **omp_get_team_num** function.

Example

```
#include <stdlib.h>
#include <assert.h>
#include <omp.h>
int foo();
int main()
{
 int res = 0, n = 0;
 #pragma omp target teams num_teams(foo()) map(res, n) reduction(+:res)
    {
    res = omp_get_team_num();
    if (omp_get_team_num() == 0)
```

```
        n = omp_get_num_teams();
     }
  Assert (res == (n*(n-1))/2);  // Sum of first n-1 natural numbers
}
```

In the preceding example, foo() is evaluated on the host and does not need to be specified using the **declare target** directive. The example performs a teams reduction on the mapped variable res. Because only the master thread of each team executes the team region, the team-private reduction variable res is safely updated to the team number of the thread. Teams are numbered from zero to the number of teams in the current teams region, which can be obtained at run time by calling the **omp_get_num_teams** function. Finally, to safely update n with the actual number of teams, only one master thread should perform the write.

#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

▶▶—#—pragma—omp threadprivate—┬—(identifier)—┬——————————————————▶◀
                              │ ┌—————,————┐│
                              └——┘   └┘

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.

Combined constructs

Combined constructs are shortcuts for specifying one construct immediately nested inside another construct. Specifying a combined construct is semantically identical to specifying the first construct that encloses an instance of the second construct and no other statements.

If you specify a clause that is permitted on both of the individual constructs, the clause applies to one or both constructs.

You can use the following combined constructs:
- **omp parallel for** [1]
- **omp parallel for simd**
- **omp parallel sections**
- **omp parallel workshare**
- **omp target parallel**
- **omp target parallel for**
- **omp target parallel for simd**
- **omp target simd**
- **omp target teams**
- **omp target teams distribute**
- **omp target teams distribute parallel for**
- **omp target teams distribute parallel for simd**
- **omp target teams distribute simd**
- **omp teams distribute**
- **omp teams distribute parallel for**
- **omp teams distribute parallel for simd**
- **omp teams distribute simd**

Note:
1. The **nowait** clause is not supported.

Chapter 6. Compiler commands reference

This section introduces the commands that are included with XL C/C++.

cleanpdf

Purpose

Removes all PDF files or the specified PDF files, including PDF files with process ID suffixes. Removing profiling information reduces runtime overhead if you change the program and then go through the PDF process again.

Syntax

```
►►──cleanpdf──┬───────┬──┬────┬──┬───────────────┬──────────────►◄
              └─pdfdir─┘  └─-u─┘  └─-f──pdfname───┘
```

Parameters

pdfdir Specifies the directory that contains the PDF files to be removed. If *pdfdir* is not specified, the directory is set by the PDFDIR environment variable; if PDFDIR is not set, the directory is the current directory.

-f *pdfname*
Specifies the name of the PDF file to be removed. If **-f** *pdfname* is not specified, **.<*output_name*>_pdf** is removed by default, where *<output_name>* is the name of the output file that is generated when you compile your program with **-qpdf1**.

-u If **-f** *pdfname* is specified, in addition to the file removed by **-f**, files with the naming convention *pdfname.<pid>*, if applicable, are also removed. *<pid>* is the ID of the running process in the PDF training step.

If **-f** *pdfname* is not specified, removes the default PDF file **.<*output_name*>_pdf**. If applicable, files with the default naming convention **.<*output_name*>_pdf.<*pid*>** are also removed.

Usage

Run **cleanpdf** only when you finish the PDF process for a particular application. Otherwise, if you want to resume by using PDF process with that application, you must compile all of the files again with **-qpdf1**.

You can find **cleanpdf** in **/opt/ibm/xlC/13.1.6/bin/**.

Related information
- "-qpdf1, -qpdf2" on page 180
- "Profile-directed feedback" in the *XL C/C++ Optimization and Programming Guide*

genhtml

Purpose

Displays the HTML version of an XML report that has already been generated.

Usage

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 suffix 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
- "-qlistfmt" on page 171

mergepdf

Purpose

Merges two or more PDF files into a single PDF file.

Syntax
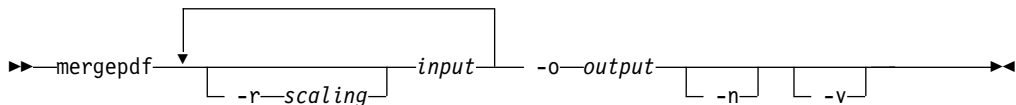
Parameters

-r *scaling*
Specifies the scaling ratio for the PDF file. This value must be greater than zero and can be either an integer or a floating-point value. If not specified, a ratio of 1.0 is assumed.

input Specifies the name of a PDF input file, or a directory that contains PDF files.

-o *output*
Specifies the name of the PDF output file, or a directory to which the merged output is written.

-n Specifies that PDF files do not get normalized.

-v Specifies verbose mode, and causes internal and user-specified scaling ratios to be displayed to standard output.

Usage

By default, **mergepdf** normalizes the files in such a way that every profile has the same overall weighting, and individual counters are scaled accordingly. This procedure is done before applying the user-specified ratio (with **-r**). When **-n** is specified, no normalization occurs. If neither **-n** nor **-r** is specified, the PDF files are not scaled at all.

You can find **mergepdf** in /opt/ibm/xlC/13.1.6/bin/.

Example

If you have several PDF files, use the **mergepdf** command to combine these PDF
files into one PDF file. For example, if you produce three PDF files that represent
usage patterns that occur 53%, 32%, and 15% of the time respectively, you can use
this command:

```
mergepdf -r 53 file_path1  -r 32 file_path2  -r 15 file_path3 -o file_path4
```

where *file_path1*, *file_path2*, and *file_path3* specify the directories and names of the
PDF files that are to be merged, and *file_path4* specifies the directory and name of
the output PDF file.

Related information
- "-qpdf1, -qpdf2" on page 180
- "Profile-directed feedback" in the *XL C/C++ Optimization and Programming Guide*

showpdf

Purpose

Displays part of the profiling information written to PDF and PDF map files. To
use this command, you must first compile your program with the **-qpdf1** option.

Syntax

```
►►─showpdf─┬──────────┬─┬──────────────────┬─┬──────────────────┬─┬───────┬─►◄
           └─pdfdir─┘   └─-f─pdfname─┘       └─-m─pdfmapdir─┘     └─-xml─┘
```

Parameters

pdfdir

Is the directory that contains the profile-directed feedback (PDF) file. If the
PDFDIR environment variable is not changed after the PDF1 step, the PDF
map file is also contained in this directory. If this parameter is not specified,
the compiler uses the value of the PDFDIR environment variable as the name
of the directory.

pdfname

Is the name of the PDF file. If this parameter is not specified, the compiler uses
.*<output_name>*_pdf as the name of the PDF file by default, where
<output_name> is the name of the output file that is generated when you
compile your program with **-qpdf1**.

pdfmapdir

Is the directory that contains the PDF map file. If this parameter is not
specified, the compiler uses the value of the PDFDIR environment variable as
the name of the directory.

-xml

Determines the display format of the PDF information. If this parameter is
specified, the PDF information is displayed in XML format; otherwise, it is
displayed in text format. Because value profiling and cache-miss profiling
information can be displayed only in XML format, the PDF report in XML
format contains more information than the report in text format.

Usage

A PDF map file that contains static information is generated during the PDF1 step and a PDF file is generated during the execution of the resulting application. With the **showpdf** command, you can view the following types of profiling information that is gathered from your application:

- Block-counter profiling
- Call-counter profiling
- Value profiling
- Cache-miss profiling, if you specified the **-qpdf1=level=2** option during the PDF1 step.

The **showpdf** command accepts only PDF files that are in binary format and needs both the PDF and PDF map files to display PDF information. You can view the first two types of profiling information in either text or XML format. However, you can view value profiling and cache-miss profiling information only in XML format.

If the PDFDIR environment variable is changed between the PDF1 step and the execution of the resulting application, the PDF and PDF map files are generated in separate directories. In this case, you must specify the directories for both of these files to the **showpdf** command.

You can find **showpdf** in `/opt/ibm/xlC/13.1.6/bin/`.

Example

The following example shows how to use the **showpdf** command to view the profiling information for a Hello World application.

The source for the program file `hello.c` is as follows:

```
#include <stdio.h>
void HelloWorld()
{
   printf("Hello World");
}
main()
{
   HelloWorld();
   return 0;
}
```

1. Compile the source file.

   ```
   xlc -qpdf1 -O hello.c
   ```

2. Run the resulting executable program **a.out** with a typical data set or several typical data sets.

3. If you want to view the profiling information for the executable file in text format, run the **showpdf** command without any parameters.

   ```
   showpdf
   ```

   The result is as follows:

   ```
   HelloWorld(67):  1 (hello.c)

   Call Counters:
   4 | 1  printf(69)

   Call coverage = 100% ( 1/1 )

   Block Counters:
   2-4 | 1
   ```

```
5 |
5 | 1

Block coverage = 100% ( 2/2 )


------------------------------------
main(68):  1 (hello.c)

Call Counters:
8 | 1  HelloWorld(67)

Call coverage = 100% ( 1/1 )

Block Counters:
6-9 | 1
10 |

Block coverage = 100% ( 1/1 )

Total Call coverage = 100% ( 2/2 )
Total Block coverage = 100% ( 3/3 )
```

If you want to view the profiling information in XML format, run the **showpdf** command with the **-xml** parameter.

```
showpdf -xml
```

The result is as follows:

```
<?xml version="1.0" encoding="UTF-8" ?>
- <XLTransformationReport xmlns="http://www.ibm.com/2010/04/CompilerTransformation" version="1.0">
  - <CompilationStep name="showpdf">
    - <ProgramHierarchy>
      - <FileList>
        - <File id="1" name="hello.c">
          - <RegionList>
              <Region id="67" name="HelloWorld" startLineNumber="2" />
              <Region id="68" name="main" startLineNumber="6" />
            </RegionList>
          </File>
        </FileList>
      </ProgramHierarchy>
      <TransformationHierarchy />
    - <ProfilingReports>
      - <BlockCounterList>
        - <BlockCounter regionId="67" execCount="1" coveredBlock="2" totalBlock="2">
          - <BlockList>
              <Block index="3" execCount="1" startLineNumber="2" endLineNumber="4" />
              <Block index="2" execCount="0" startLineNumber="5" endLineNumber="5" />
              <Block index="4" execCount="1" startLineNumber="5" endLineNumber="5" />
            </BlockList>
          </BlockCounter>
        - <BlockCounter regionId="68" execCount="1" coveredBlock="1" totalBlock="1">
          - <BlockList>
              <Block index="3" execCount="1" startLineNumber="6" endLineNumber="9" />
              <Block index="2" execCount="0" startLineNumber="10" endLineNumber="10" />
            </BlockList>
          </BlockCounter>
        </BlockCounterList>
      - <CallCounterList>
        - <CallCounter regionId="67" execCount="1" coveredCall="0" totalCall="0">
          - <CallList>
              <Call name="printf" execCount="1" lineNumber="4" />
            </CallList>
          </CallCounter>
        - <CallCounter regionId="68" execCount="1" coveredCall="0" totalCall="0">
          - <CallList>
              <Call name="HelloWorld" execCount="1" lineNumber="8" />
            </CallList>
          </CallCounter>
        </CallCounterList>
        <ValueProfileList />
        <CacheMissList />
      </ProfilingReports>
    </CompilationStep>
  </XLTransformationReport>
```

Related information

- "-qpdf1, -qpdf2" on page 180
- "-qshowpdf" on page 197
- "Profile-directed feedback" in the *XL C/C++ Optimization and Programming Guide*

Chapter 7. Macros reference

This section provides information about the macros that are supported by XL C/C++.

The macros that are listed in "Compiler predefined macros" have compiler-predefined values. These predefined macros are used to conditionally compile code for specific compilers, specific versions of compilers, specific environments, and specific language features. Some predefined macros are protected, which means that the compiler will issue a warning message if you try to undefine or redefine them. Some predefined macros are unprotected and can be undefined or redefined without warning.

The compiler also provides many other macros that you can use in your programs. For example, XL C/C++ supports function-like macros with a variable number of arguments, as a language extension for compatibility with C and as part of C++11. The "Other macros" on page 314 section lists some of the compiler-supplied macros that do not have predefined values.

Compiler predefined macros

Predefined macros can be used to conditionally compile code for specific compilers, specific versions of compilers, specific environments, and specific language features.

Predefined macros fall into several categories:
- "General macros"
- "Macros related to the platform" on page 308
- "Macros related to compiler features" on page 308

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 30. 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.

Table 30. General predefined macros (continued)

Predefined macro name	Description	Predefined value
__SIZE_TYPE__	Indicates the underlying type of `size_t` on the current platform. Not protected.	`unsigned long`
__TIME__	Indicates the time that the source file was preprocessed.	A character string containing the time when the source file was preprocessed.
__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 to identify the XL C/C++ compiler for cross-compilation

Most of the macros related to the XL C/C++ compiler are predefined and protected, which means that the compiler will issue a warning if you try to undefine or redefine them. You can use the **-dM (-qshowmacros) -E** compiler options to view the values of the predefined macros.

Table 31. Compiler-related predefined macros

Predefined macro name	Description	Predefined value
▶ C __IBMC__ [1]	Indicates the level of the XL C compiler.	An integer in format *VRM*, where: *V* Represents the version number *R* Represents the release number *M* Represents the modification number
▶ C++ __IBMCPP__ [1]	Indicates the level of the XL C++ compiler.	An integer in format *VRM*, where: *V* Represents the version number *R* Represents the release number *M* Represents the modification number
__xlC__ [1]	Indicates the VR level of the XL C and XL C++ compilers in hexadecimal format. The XL C compiler predefines this macro.	A 4-digit hexadecimal integer in format 0x*VVRR*, where: *V* Represents the version number *R* Represents the release number

Table 31. Compiler-related predefined macros (continued)

Predefined macro name	Description	Predefined value
__xlC_ver__[1]	Indicates the MF level of the XL C and XL C++ compilers in hexadecimal format. The XL C compiler predefines this macro.	An 8-digit hexadecimal integer in format 0x*0000MMFF*, where: *M* Represents the modification number *F* Represents the fix level
▶ C __xlc__[1]	Indicates the level of the XL C compiler.	A string in 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
__clang__	Indicates that Clang compiler is used.	1
__clang_major__	Indicates the major version number of the Clang compiler.	4
__clang_minor__	Indicates the minor version number of the Clang compiler.	0
__clang_patchlevel__	Indicates the patch level number of the Clang compiler.	1
__clang_version__	Indicates the full version of the Clang compiler.	4.0.1 (tags/RELEASE_401/final)
__ibmxl__	Indicates the XL C/C++ compiler is being used.	1
__ibmxl_vrm__	Indicates the VRM level of the XL C/C++ compiler using a single integer for sorting purposes.	A hexadecimal integer whose value is as follows: `(((__ibmxl_version__) << 24)	\` ` ((__ibmxl_release__) << 16)	\` ` ((__ibmxl_modification__) << 8) \` `)`
__ibmxl_version__	Indicates the version number of the XL C/C++ compiler.	An integer that represents the version number
__ibmxl_release__	Indicates the release number of the XL C/C++ compiler.	An integer that represents the release number
__ibmxl_modification__	Indicates the modification number of the XL C/C++ compiler.	An integer that represents the modification number
__ibmxl_ptf_fix_level__	Indicates the PTF fix level of the XL C/C++ compiler.	An integer that represents the fix number
__llvm__	Indicates that an LLVM backend is used.	1

Note:

1. This macro is not defined when the default **-qnoxlcompatmacros** option is in effect. To migrate programs from IBM XL C/C++ for Linux V13.1 or earlier for big endian distributions or IBM XL C/C++ for AIX to release versions starting from IBM XL C/C++ for Linux, V13.1.6 for little endian distributions, you must use the **-qxlcompatmacros** option to define this macro.

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 32. Platform-related predefined macros

Predefined macro name	Description	Predefined value	Predefined under the following conditions
__ELF__	Indicates that the ELF object model is in effect.	1	Always predefined for the Linux platform.
▶ **C++** __GXX_WEAK__	Indicates that weak symbols are supported (used for template instantiation by the linker).	1	Always predefined.
__HOS_LINUX__	Indicates that the host operating system is Linux. Protected.	1	Always predefined for all Linux platforms.
__linux, __linux__, linux, __gnu_linux__	Indicates that the platform is Linux.	1	Always predefined for all Linux platforms.
_LITTLE_ENDIAN, __LITTLE_ENDIAN__	Indicates that the platform is little-endian (that is, the most significant byte is stored at the memory location with the highest address).	1	Always predefined.
_LP64, __LP64__	Indicates that the target platform uses 64-bit `long int` and pointer types, and a 32-bit `int` type.	1	Predefined when the target platform uses 64-bit `long int` and pointer types, and 32-bit a `int` type.
__POWERPC__	Indicates that the target is a Power architecture.	1	Predefined when the target is a Power architecture.
__PPC__	Indicates that the target is a Power architecture.	1	Predefined when the target is a Power architecture.
__PPC64__	Indicates that the target is a Power architecture and that 64-bit compilation mode is enabled.	1	Always predefined.
__THW_PPC__	Indicates that the target is a Power architecture.	1	Predefined when the target is a Power architecture.
__TOS_LINUX__	Indicates that the target operating system is Linux.	1	Predefined when the target OS is Linux.
__unix, __unix__, unix	Indicates that the operating system is a variety of UNIX.	1	Always predefined.

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, which means that 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 310
- "Macros related to language levels" on page 311

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 33. 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	Always predefined.
__ALTIVEC__	Indicates support for vector data types. (unprotected)	1	-qaltivec
_CHAR_SIGNED, __CHAR_SIGNED__	Indicates that the default character type is `signed char`.	1	-fsigned-char (-qchars=signed)
_CHAR_UNSIGNED, __CHAR_UNSIGNED__	Indicates that the default character type is `unsigned char`.	1	-funsigned-char (-qchars=unsigned)
C++ __EXCEPTIONS	Indicates that C++ exception handling is enabled.	1	-fexceptions (-qeh)
__GXX_RTTI	Indicates that runtime type identification (RTTI) information is enabled.	1	-qrtti, -fno-rtti (-qnortti)
C _IBMSMP	Indicates that IBM SMP directives are recognized.	1	-qsmp
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 33. 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
__LONGDOUBLE128, __LONG_DOUBLE_128__	Indicates that the size of a `long double` type is 128 bits.	1	Always predefined.
__OPTIMIZE__	Indicates the level of optimization in effect.	2	-O | -O2
		3	-O3
		4	-O4 | -O5
__OPTIMIZE_SIZE__	Indicates that optimization for code size is in effect.	1	-O | -O2 | -O3 | -O4 | -O5 and -qcompact
__RTTI_ALL__	Indicates that runtime type identification (RTTI) information for all operators is enabled.	1	-qrtti
C++ __NO_RTTI__	Indicates that runtime type identification (RTTI) information is disabled.	1	-fno-rtti (-qnortti)
__VEC__	Indicates support for vector data types.	10206	-qaltivec
__VEC_ELEMENT_REG_ORDER__	Indicates the vector element order used in vector registers.	• __ORDER_LITTLE_ENDIAN__ when -maltivec=le (-qaltivec=le) is in effect • __ORDER_BIG_ENDIAN__ when -maltivec=be (-qaltivec=be) is in effect	-qaltivec

Macros related to architecture settings

The following macros can be tested for target architecture settings. All of these macros are predefined to a value of 1 by a **-mcpu** compiler option setting, or any other compiler option that implies that setting. If the **-mcpu** suboption does not enabling the feature, then the macro is undefined.

Table 34. -mcpu-related macros

Macro name	Description	Predefined by the following -mcpu suboptions
_ARCH_PPC	Indicates that the application is targeted to run on any Power processor.	Defined for all -mcpu suboptions except auto.
_ARCH_PPC64	Indicates that the application is targeted to run on Power processors with 64-bit support.	pwr8 | pwr9
_ARCH_PPCGR	Indicates that the application is targeted to run on Power processors with graphics support.	pwr8 | pwr9
_ARCH_PWR4	Indicates that the application is targeted to run on POWER4 or higher processors.	pwr8 | pwr9
_ARCH_PWR5	Indicates that the application is targeted to run on POWER5 or higher processors.	pwr8 | pwr9
_ARCH_PWR5X	Indicates that the application is targeted to run on POWER5+ or higher processors.	pwr8 | pwr9
_ARCH_PWR6	Indicates that the application is targeted to run on POWER6® or higher processors.	pwr8 | pwr9
_ARCH_PWR7	Indicates that the application is targeted to run on POWER7®, POWER7+™ or higher processors.	pwr8 | pwr9
_ARCH_PWR8	Indicates that the application is targeted to run on POWER8 processors, or higher processors.	pwr8 | pwr9
_ARCH_PWR9	Indicates that the application is targeted to run on POWER9 processors.	pwr9

Related information

- "-mcpu (-qarch)" on page 131

Macros related to language levels

The following macros except ▶ **C++** __cplusplus, __STDC__ **C++** ◀ , and ▶ **C** __STDC_VERSION__ **C** ◀ are predefined to a value of 1 by a specific language level, represented by a suboption of the **-std (-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 and the C and C++ language standards*.

Table 35. 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.
▶ **C++** __cplusplus	The numeric value that indicates the supported language standard as defined by that specific standard.	The format is *yyyymm*L. (For example, the format is 199901L for C99.)

Table 35. Predefined macros for language features (continued)

Predefined macro name	Description	Predefined when the following language level is in effect
C++ __IBMCPP_COMPLEX_INIT	Indicates support for the initialization of complex types: float _Complex, double _Complex, and long double _Complex.	extended	extended0x
__STDC__	Indicates that the compiler conforms to the ANSI/ISO C standard.	**C** Predefined to 1 if ANSI/ISO C standard conformance is in effect. **C++** Explicitly defined to 0.
__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** stdc11	extc1x	stdc99	extc99 **C++** extended0x| extended1y
C11 __STDC_NO_ATOMICS__	Indicates that the implementation does not have the full support of the atomics feature.	stdc11	extc1x
C11 __STDC_NO_THREADS__	Indicates that the implementation does not have the full support of the threads feature.	stdc11	extc1x
C __STDC_VERSION__	Indicates the version of ANSI/ISO C standard which the compiler conforms to.	The format is *yyyymm*L. (For example, the format is 199901L for C99.)

Unsupported macros from other XL compilers

The following macros, which might be supported by other XL compilers, are unsupported in IBM XL C/C++ for Linux, V13.1.6. You can specify the **-Wunsupported-xl-macro** option to check whether any unsupported macro is used; if an unsupported macro is used, the compiler issues a warning message.

You might want to edit your source code to remove references of the unsupported macros during compiler migration.

Table 36. Unsupported macros that are related to the platform

_BIG_ENDIAN, __BIG_ENDIAN__
_ILP32, __ILP32__
__THW_370__
__THW_BIG_ENDIAN__

Table 37. Unsupported macros related to compiler option settings

__LONGDOUBLE64
__IBM_GCC_ASM
__IBM_STDCPP_ASM
__TEMPINC__

Table 38. Unsupported macros related to architecture settings

_ARCH_PWR6E

Table 39. Unsupported macros related to language levels

__C99_BOOL	__IBM_DOLLAR_IN_ID
__C99_COMPLEX	__IBM_EXTENSION_KEYWORD
__C99_COMPOUND_LITERAL	__IBM_GCC__INLINE__
__C99_CPLUSCMT	__IBM_GENERALIZED_LVALUE
__C99_DESIGNATED_INITIALIZER	__IBM_INCLUDE_NEXT
__C99_DUP_TYPE_QUALIFIER	__IBM_LABEL_VALUE
__C99_EMPTY_MACRO_ARGUMENTS	__IBM_LOCAL_LABEL
__C99_FLEXIBLE_ARRAY_MEMBER	__IBM_MACRO_WITH_VA_ARGS
__C99_FUNC__	__IBM_NESTED_FUNCTION
__C99_HEX_FLOAT_CONST	__IBM_PP_PREDICATE
__C99_INLINE	__IBM_PP_WARNING
__C99_LLONG	__IBM_REGISTER_VARS
__C99_MACRO_WITH_VA_ARGS	__IBM__TYPEOF__
__C99_MAX_LINE_NUMBER	__IBMC_COMPLEX_INIT
__C99_MIXED_DECL_AND_CODE	__IBMC_GENERIC
__C99_MIXED_STRING_CONCAT	__IBMC_NORETURN
__C99_NON_LVALUE_ARRAY_SUB	__IBMC_STATIC_ASSERT
__C99_NON_CONST_AGGR_INITIALIZER	__IBMCPP_AUTO_TYPEDEDUCTION
__C99_PRAGMA_OPERATOR	__IBMCPP_C99_LONG_LONG
__C99_REQUIRE_FUNC_DECL	__IBMCPP_C99_PREPROCESSOR
__C99_RESTRICT	__IBMCPP_CONSTEXPR
__C99_STATIC_ARRAY_SIZE	__IBMCPP_DECLTYPE
__C99_STD_PRAGMAS	__IBMCPP_DELEGATING_CTORS
__C99_TGMATH	__IBMCPP_EXPLICIT_CONVERSION_OPERATORS
__C99_UCN	__IBMCPP_EXTENDED_FRIEND
__C99_VAR_LEN_ARRAY	__IBMCPP_EXTERN_TEMPLATE
__C99_VARIABLE_LENGTH_ARRAY	__IBMCPP_INLINE_NAMESPACE
__DIGRAPHS__	__IBMCPP_REFERENCE_COLLAPSING
__EXTENDED__	__IBMCPP_RIGHT_ANGLE_BRACKET
__IBM__ALIGN	__IBMCPP_RVALUE_REFERENCES
__IBM__ALIGNOF__	__IBMCPP_SCOPED_ENUM
__IBM_ALIGNOF__	__IBMCPP_STATIC_ASSERT
__IBM_ATTRIBUTES	__IBMCPP_UNIFORM_INIT
__IBM_COMPUTED_GOTO	__IBMCPP_VARIADIC_TEMPLATES
	_LONG_LONG

Other macros

These macros are supported by XL C/C++. They do not have predefined values.

Table 40. Other macros

Macro name	Description	Example
__has_include	Checks for the existence of a given include file. You must use this macro as expressions in #if or #elif preprocessing directives. The macro evaluates to 1 if the given include file is found in the include paths; otherwise, it evaluates to 0.	__has_include("*include_file_name*.h") __has_include(<*include_file_name*.h>)
__has_include_next	Checks for a second instance of a given include file. You must use this macro as expressions in #if or #elif preprocessing directives. You can use this macro only in headers. If the macro is used in the top-level compilation file, the compiler will issue a warning message. The compiler will also issue a warning message if an absolute path is used in the file argument. The macro evaluates to 1 if a second instance of the given include file is found in the include paths; otherwise, it evaluates to 0.	__has_include_next("*include_file_name*.h") __has_include_next(<*include_file_name*.h>)

Chapter 8. Compiler built-in functions

A built-in function is a coding extension to C and C++ that allows a programmer to use the syntax of C function calls and C variables to access the instruction set of the processor of the compiling machine. IBM Power architectures have special instructions that enable the development of highly optimized applications. Access to some Power instructions cannot be generated using the standard constructs of the C and C++ languages. Other instructions can be generated through standard constructs, but using built-in functions allows exact control of the generated code. Inline assembly language programming, which uses these instructions directly, is fully supported starting from XL C/C++, V12.1. 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. **C++** ◀

The following sections describe the available built-in functions for the Linux platform.

Fixed-point built-in functions

Fixed-point built-in functions are grouped into the following categories:
* "Absolute value functions"
* "Assert functions" on page 316
* "Count zero functions" on page 319
* "Load functions" on page 321
* "Multiply functions" on page 322
* "Population count functions" on page 323
* "Rotate functions" on page 324
* "Store functions" on page 326
* "Trap functions" on page 326

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);

Add extended functions

__builtin_addex
Purpose

Adds Extended using Overflow Bit.

This built-in function is valid only when -qarch(-mcpu) is set to utilize the POWER9 technology.

Prototype

signed long long __builtin_addex(signed long long, signed long long, const int);

unsigned long long __builtin_addex(unsigned long long, unsigned long long, const int);

Usage

The third parameter is 0. The parameter equal to 1, 2, or 3 is reserved.

The result is the sum of the first parameter, the second parameter, and the Overflow Bit.

Assert functions

__assert1, __assert2
Purpose

Generates trap instructions.

Prototype

int __assert1 (int a, int b, int c);

void __assert2 (int a);

Usage

The __assert1 function compares parameter a and b using the condition that is specified by c. Parameter c must be a literal or a compiler-time constant that is in the range from 1 to 31, inclusive. Refer to the following table for the mapping relations between the comparison condition and the value of c.

Table 41. Comparison condition specified by c

Value	Comparison condition
16	<
8	>
4	==
2	<[1]
1	>[1]

Table 41. Comparison condition specified by c *(continued)*

Value	Comparison condition
Notes:	
1. The values of a and b are converted to unsigned integers before performing the comparison.	
2. The values in the table can be ORed to get more comparison conditions; for example, 24 indicates != for signed integers.	

Comparison at compile time

If the comparison can be evaluated at compile time, the compiler performs the comparison at compile time and has the following behaviors:

- If the comparison result is true, an unconditional trap instruction is generated, a trap occurs, and a SIGTRAP signal is generated. If the SIGTRAP signal handler exists, the signal handler is executed. If the SIGTRAP signal handler does not exist, the program is terminated with a SIGTRAP exit code.
- If the comparison result is false, no trap instruction is generated, no trap occurs, and no SIGTRAP signal is generated.

Comparison at run time

If the comparison cannot be evaluated at compile time, the compiler generates a conditional trap instruction to check the comparison at run time and has the following behaviors:

- If the comparison result is true, a trap occurs, and a SIGTRAP signal is generated. If the SIGTRAP signal handler exists, the signal handler is executed. If the SIGTRAP signal handler does not exist, the program is terminated with a SIGTRAP exit code.
- If the comparison result is false, no trap occurs, and no SIGTRAP signal is generated.

The compiler might remove the trap instruction during the optimization. To avoid this issue, use the result of __assert1 as the input of the__assert2 function to create a dependency between these two functions. The__assert2 function acts as a compile-time trap anchor that prevents the trap or trap immediate instruction that is generated for __assert1 from being discarded during compiler optimization unless the instruction is unreachable.

Bit permutation functions

__bpermd
Purpose

Byte Permute Doubleword

Returns the result of a bit permutation operation.

Prototype

long long __bpermd (long long bit_selector, long long source);

Usage

Eight bits are returned, each corresponding to a bit within source, and were selected by a byte of bit_selector. If byte i of bit_selector is less than 64, the permuted bit i is set to the bit of source specified by byte i of bit_selector; otherwise, the permuted bit i is set to 0. The permuted bits are placed in the least-significant byte of the result value and the remaining bits are filled with 0s.

Comparison functions

__builtin_cmpeqb
Purpose

Compares the corresponding bytes of the given parameters and returns the result.

This built-in function is valid only when `-qarch(-mcpu)` is set to utilize the POWER9 technology.

Prototype

signed long long __builtin_cmpeqb (signed long long a, signed long long b);

Usage

If the rightmost byte of a is equal to any byte of b, the result is set to 1 ; otherwise, the result is set to 0.

__builtin_cmprb
Purpose

Compares the ranged byte.

This built-in function is valid only when `-qarch(-mcpu)` is set to utilize the POWER9 technology.

Prototype

int __builtin_cmprb(const int a, int b, int c)

Usage
- If a is 0, the result is set by the following rules:
    - If the rightmost byte of b is in the range from the rightmost byte of c to the second rightmost bytes of c, the result is 1.
    - Otherwise, the result is 0.
- If a is 1, the result is set by the following rules:
    - If the rightmost byte of b is in the range from the rightmost byte to the second rightmost byte of either halfword of c, the result is 1.
    - Otherwise, the result is 0.

Notes:
- a can only be 0 or 1.
- The rightmost byte is the low bound of the range and second rightmost byte is the high bound of the range.

__builtin_setb
Purpose

Sets boolean extension.

This built-in function is valid only when `-qarch(-mcpu)` is set to utilize the POWER9 technology.

Prototype

> long long __builtin_setb (signed long long a, signed long long b);

Usage

The function compares the two parameters and returns the result:
- If a is smaller than b, the result is -1.
- If a is larger than b, the result is 1.
- If a is equal to b, the result is 0.

__cmpb
Purpose

Compare Bytes

Compares each of the eight bytes of *source1* with the corresponding byte of *source2*. If byte *i* of *source1* and byte *i* of *source2* are equal, 0xFF is placed in the corresponding byte of the result; otherwise, 0x00 is placed in the corresponding byte of the result.

Prototype

> long long __cmpb (long long source1, long long source2);

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);

Division functions

__divde
Purpose

Divide Doubleword Extended

Returns the result of a doubleword extended division. The result has a value equal to *dividend/divisor*.

Prototype

> long long __divde (long long dividend, long long divisor);

Usage

If the result of the division is larger than 32 bits or if the divisor is 0, the return value of the function is undefined.

__divdeu
Purpose

Divide Doubleword Extended Unsigned

Returns the result of a double word extended unsigned division. The result has a value equal to *dividend/divisor*.

Prototype

> unsigned long long __divdeu (unsigned long long dividend, unsigned long long divisor);

Usage

If the result of the division is larger than 32 bits or if the divisor is 0, the return value of the function is undefined.

__divwe
Purpose

Divide Word Extended

Returns the result of a word extended division. The result has a value equal to *dividend/divisor*.

Prototype

> int __divwe(int dividend, int divisor);

Usage

If the divisor is 0, the return value of the function is undefined.

__divweu
Purpose

Divide Word Extended Unsigned

Returns the result of a word extended unsigned division. The result has a value equal to *dividend/divisor*.

Prototype

> unsigned int __divweu(unsigned int dividend, unsigned int divisor);

Usage

If the divisor is 0, the return value of the function is undefined.

Load functions

__load2r
Purpose

Load Halfword Byte Reversed

Performs a two-byte byte-reversed load from the given address.

Prototype

> unsigned short __load2r (unsigned short*);

__load4r
Purpose

Load Word Byte Reversed

Performs a four-byte byte-reversed load from the given address.

Prototype

> unsigned int __load4r (unsigned int*);

__load8r
Purpose

Load with Byte Reversal (8-byte integer)

Performs an eight-byte byte-reversed load from the given address.

Prototype

> unsigned long long __load8r (unsigned long long * address);

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);

Multiply-add functions

__builtin_maddhd
Purpose

Fixed-point Multiply-Add signed high doubleword.

Multiplies the first two arguments, adds the third argument, and returns the high doubleword of the result.

This built-in function is valid only when -qarch(-mcpu) is set to utilize the POWER9 technology.

Prototype

signed long long __builtin_maddhd (signed long long, signed long long, signed long long);

__builtin_maddhdu
Purpose

Fixed-point Multiply-Add high doubleword unsigned.

Multiplies the first two arguments, adds the third argument, and returns the high doubleword of the result.

This built-in function is valid only when -qarch(-mcpu) is set to utilize the POWER9 technology.

Prototype

> unsigned long long __builtin_maddhdu (unsigned long long, unsigned long long, unsigned long long);

__builtin_maddld
Purpose

Fixed-point Multiply-Add low doubleword.

Multiplies the first two arguments, adds the third argument, and returns the low doubleword of the result.

This built-in function is valid only when `-qarch(-mcpu)` is set to utilize the POWER9 technology.

Prototype

> signed long long __builtin_maddld (signed long long, signed long long, signed long long);

> unsigned long long __builtin_maddld (unsigned long long, unsigned long long, unsigned long long);

Population count functions

__popcnt4, __popcnt8
Purpose

Population Count, 4-byte or 8-byte integer

Returns the number of bits set for a 32-bit or 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.

Random number functions

__builtin_darn, __builtin_darn_32, __builtin_darn_raw
Purpose

Delivers a random number.

This built-in function is valid only when -qarch(-mcpu) is set to utilize the POWER9 technology.

Prototype

long long __builtin_darn (void);

int __builtin_darn_32 (void);

long long __builtin_darn_raw (void);

Usage

- __builtin_darn returns a random number in the range 0 to 0xFFFFFFFF_FFFFFFFE. 0xFFFFFFFF_FFFFFFFF indicates an error condition. The result has been processed by hardware to reduce bias.
- __builtin_darn_32 returns a random number in the range 0 to 0xFFFFFFFF. The result has been processed by hardware to reduce bias.
- __builtin_darn_raw returns a random number in the range 0 to 0xFFFFFFFF_FFFFFFFE. 0xFFFFFFFF_FFFFFFFF indicates an error condition.

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
Purpose

Store with Byte-Reversal (two-byte integer).

Takes the loaded two-byte integer value and performs a byte-reversed store operation.

Prototype

> void __store2r (unsigned short, unsigned short*);

__store4r
Purpose

Store with Byte-Reversal (four-byte integer).

Takes the loaded four-byte integer value and performs a byte-reversed store operation.

Prototype

> void __store4r (unsigned int, unsigned int*);

__store8r
Purpose

Store with Byte-Reversal (eight-byte integer).

Takes the loaded eight-byte integer value and performs a byte-reversed store operation.

Prototype

> void __store8r (unsigned long long source, unsigned long long * address);

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" on page 315
- "Conversion functions" on page 329
- "FPSCR functions" on page 332
- "Multiply-add/subtract functions" on page 334
- "Reciprocal estimate functions" on page 335
- "Rounding functions" on page 335
- "Select functions" on page 337
- "Square root functions" on page 337
- "Software division functions" on page 338

Absolute value functions

__fnabss
Purpose

Floating Absolute Value Single

Returns the absolute value of the argument.

Prototype

> float __fnabss (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);

Comparison functions

__builtin_compare_exp_uo, __builtin_compare_exp_lt, __builtin_compare_exp_eq, __builtin_compare_exp_gt
Purpose

Compares the exponents of two parameters.

This built-in function is valid only when -qarch(-mcpu) is set to utilize the POWER9 technology.

Prototype

int __builtin_compare_exp_uo (double, double);

int __builtin_compare_exp_lt (double, double);

int __builtin_compare_exp_eq (double, double);

int __builtin_compare_exp_gt (double, double);

Usage
1. If either of the parameters of __builtin_compare_exp_uo is NaN, __builtin_compare_exp_uo returns 1 and the following built-in functions return 0: __builtin_compare_exp_lt, __builtin_compare_exp_eq, and __builtin_compare_exp_gt.
2. If neither of the parameters is NaN, the result of __builtin_compare_exp_uo is 0. In this case, the following built-in functions show the comparison result of the exponents of two parameters:
   - If the exponent of the first parameter is smaller than the exponent of the second, the result of __builtin_compare_exp_lt is 1; otherwise, the result is 0.
   - If the exponent of the first parameter is larger than the exponent of the second, the result of __builtin_compare_exp_gt is 1; otherwise, the result is 0.
   - If the exponent of the first parameter is equal to the exponent of the second, the result of __builtin_compare_exp_eq is 1; otherwise, the result is 0.

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);

__fcfud

Purpose

Floating-point Conversion from Unsigned integer Double word

Converts a 64-bit unsigned integer stored in a double into a double-precision floating-point value.

Prototype

double __fcfud(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);

__fctudz
Purpose

Floating-point Conversion to Unsigned integer Double word with rounding towards Zero

Converts a floating-point value to unsigned integer double word and rounds to zero.

Prototype

> double __fctudz(double);

Result value

The result is a double number, which is rounded to zero.

__fctuwz
Purpose

Floating-point conversion to unsigned integer word with rounding to zero

Converts a floating-point number into a 32-bit unsigned integer and rounds to zero. The conversion result is stored in a double return value. This function is intended for use with the __stfiw built-in function.

Prototype

> double __fctuwz(double);

Result value

The result is a double number. The low-order 32 bits of the result contain the unsigned int value from converting the double parameter to unsigned int, rounded to zero. The high-order 32 bits contain an undefined value.

Example

The following example demonstrates the usage of this function.

```
#include <stdio.h>

int main(){
  double result;
  int y;

  result = __fctuwz(-1.5);
  __stfiw(&y, result);
  printf("%d\n", y);           /* prints 0 */

  result = __fctuwz(1.5);
  __stfiw(&y, result);
  printf("%d\n", y);           /* prints 1 */

  return 0;
}
```

__ibm2gccldbl, __ibm2gccldbl_cmplx (IBM extension)
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.

Extract exponent functions

__builtin_extract_exp
Purpose

Returns the exponent of the given parameter.

This built-in function is valid only when -qarch(-mcpu) is set to utilize the POWER9 technology.

Prototype

> unsigned int __builtin_extract_exp (double);

Extract significand functions

__builtin_extract_sig
Purpose

Returns the significand of the given parameter.

This built-in function is valid only when `-qarch(-mcpu)` is set to utilize the POWER9 technology.

Prototype

> unsigned long long __builtin_extract_sig (double);

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 bits 32 - 63 contain the content of the FPSCR.

Prototype

> double __readflm (void);

__setflm
Purpose

Takes a double precision floating-point number and places the value of bits 32 - 63 in the FPSCR. Returns the previous content 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

Insert exponent functions

__builtin_insert_exp
Purpose

Replaces the exponent of the first parameter with the second parameter and returns the result.

This built-in function is valid only when -qarch(-mcpu) is set to utilize the POWER9 technology.

Prototype

double __builtin_insert_exp (double, unsigned long long);

Usage

The rightmost doubleword of the result is undefined.

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 337.

__fre, __fres
Purpose

Floating Reciprocal Estimate, Floating Reciprocal Estimate Single

Prototype

    double __fre (double);

    float __fres (float);

Rounding functions

__fric
Purpose

Floating-point Rounding to Integer with current rounding mode

Rounds a double-precision floating-point value to integer with the current rounding mode.

Prototype

    double __fric(double);

__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);

__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);

__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);

__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);

Select functions

__builtin_max

Purpose

Returns the value of the largest input argument.

Prototype

A __builtin_max (A1, A2, A3, ...);

Parameters

All arguments must have the same type; they must all be float, double, or long double.

__builtin_min

Purpose

Returns the value of the smallest input argument.

Prototype

A __builtin_min (A1, A2, A3, ...);

Parameters

All arguments must have the same type; they must all be float, double, or long double.

__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);

__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*);

Test data class functions

__builtin_test_data_class
Purpose

Determines the data class of the given parameter.

This built-in function is valid only when -qarch(-mcpu) is set to utilize the POWER9 technology.

Prototype

> bool __builtin_test_data_class (float a, const int b);

> bool __builtin_test_data_class (double a, const int b);

Usage

Returns the result of testing a for the conditions selected by b. The value of b is in the range 0-127. Each bit of b enables the test of a condition. You can refer to the following table for the mapping relations between testing conditions and bits of b:

Table 42. Bitmask for testing conditions

Bits of b	Test conditions
0x01	Test for -Denormal
0x02	Test for +Denormal
0x04	Test for -Zero
0x08	Test for +Zero

Table 42. Bitmask for testing conditions (continued)

Bits of b	Test conditions
0x10	Test for `-Infinity`
0x20	Test for `+Infinity`
0x40	Test for `NaN`

If any of the enabled test conditions is true, the result is set to 1. If all of the enabled test conditions are false, the result is set to 0.

Binary-coded decimal built-in functions

Binary-coded decimal (BCD) values are compressed, with each decimal digit and sign bit occupying 4 bits. Digits are ordered right-to-left in the order of significance, and the final 4 bits encode the sign. A valid encoding must have a value in the range 0 - 9 in each of its 31 digits and a value in the range 10 - 15 for the sign field.

Source operands with sign codes of 0b1010, 0b1100, 0b1110, or 0b1111 are interpreted as positive values. Source operands with sign codes of 0b1011 or 0b1101 are interpreted as negative values.

BCD arithmetic operations encode the sign of their result as follows: A value of 0b1101 indicates a negative value, while 0b1100 and 0b1111 indicate positive values or zero, depending on the value of the preferred sign (PS) bit. These built-in functions can operate on values of at most 31 digits.

BCD values are stored in memory as contiguous arrays of 1-16 bytes.

Note:
- To use the following built-in functions, you must include the `altivec.h` file:
  - BCD add and subtract functions
  - BCD comparison functions
  - BCD load and store functions
  - BCD test add and subtract for overflow functions
- To use the BCD format conversion functions, you must include the `bcd.h` file.

BCD add and subtract

The following functions are valid when **-qarch** is set to target POWER8 or higher processors:
- "__bcdadd"
-

Note: To use the built-in functions, you must include the `altivec.h` file.

__bcdadd
Purpose

Returns the result of addition on the BCD values *a* and *b*.

The sign of the result is determined as follows:
- If the result is a non-negative value and *ps* is 0, the sign is set to 0b1100 (0xC).

- If the result is a non-negative value and *ps* is 1, the sign is set to 0b1111 (0xF).
- If the result is a negative value, the sign is set to 0b1101 (0xD).

This built-in function is valid only when you include the `altivec.h` file.

Prototype

vector unsigned char __bcdadd (vector unsigned char *a*, vector unsigned char *b*, long *ps*);

Parameters

ps   A compile-time known constant.

__bcdsub
Purpose

Returns the result of subtraction on the BCD values *a* and *b*.

The sign of the result is determined as follows:
- If the result is a non-negative value and *ps* is 0, the sign is set to 0b1100 (0xC).
- If the result is a non-negative value and *ps* is 1, the sign is set to 0b1111 (0xF).
- If the result is a negative value, the sign is set to 0b1101 (0xD).

This built-in function is valid only when you include the `altivec.h` file.

Prototype

vector unsigned char __bcdsub (vector unsigned char *a*, vector unsigned char *b*, long *ps*);

Parameters

ps   A compile-time known constant.

BCD comparison

The following functions are valid when **-qarch** is set to target POWER8 or higher processors:
- "__bcdcmpeq"
- "__bcdcmpge" on page 342
- "__bcdcmpgt" on page 342
- "__bcdcmple" on page 342
- "__bcdcmplt" on page 342

Note: To use the built-in functions, you must include the `altivec.h` file.

__bcdcmpeq
Purpose

Returns 1 if the BCD value *a* is equal to *b*, or 0 otherwise.

This built-in function is valid only when you include the `altivec.h` file.

Prototype

long __bcdcmpeq (vector unsigned char *a*, vector unsigned char *b*);

__bcdcmpge
Purpose

Returns 1 if the BCD value *a* is greater than or equal to *b*, or 0 otherwise.

This built-in function is valid only when you include the `altivec.h` file.

Prototype

long __bcdcmpge (vector unsigned char *a*, vector unsigned char *b*);

__bcdcmpgt
Purpose

Returns 1 if the BCD value *a* is greater than *b*, or 0 otherwise.

This built-in function is valid only when you include the `altivec.h` file.

Prototype

long __bcdcmpgt (vector unsigned char *a*, vector unsigned char *b*);

__bcdcmple
Purpose

Returns 1 if the BCD value *a* is less than or equal to *b*, or 0 otherwise.

This built-in function is valid only when you include the `altivec.h` file.

Prototype

long __bcdcmple (vector unsigned char *a*, vector unsigned char *b*);

__bcdcmplt
Purpose

Returns 1 if the BCD value *a* is less than *b*, or 0 otherwise.

This built-in function is valid only when you include the `altivec.h` file.

Prototype

long __bcdcmplt (vector unsigned char *a*, vector unsigned char *b*);

BCD format conversion

The functions are valid when **-qarch** is set to utilize POWER9 technology.

Note: To use the built-in functions, you must include the `bcd.h` file.

__builtin_national2packed

Purpose

Transforms the national decimal value to packed decimal format.

This built-in function is valid only when -qarch(-mcpu) is set to utilize the POWER9 technology and the altivec.h file is included.

Prototype

vector unsigned char __builtin_national2packed(vector unsigned char *a*, unsigned char *b*);

Usage

Transforms a from national decimal format to packed decimal format.

The value of b can only be 0 or 1.

The sign code of the result is set according to the following rules:
- If a is positive, the sign code is set according to the following rules:
  - If b is 0, the sign code is set to 0xC.
  - If b is 1, the sign code is set to 0xF.
- If a is negative, the sign code is set to 0xD.

__builtin_packed2national

Purpose

Transforms the packed decimal value to national decimal format.

This built-in function is valid only when -qarch(-mcpu) is set to utilize the POWER9 technology and the altivec.h file is included.

Prototype

vector unsigned char __builtin_packed2national(vector unsigned char);

Usage

If the input parameter is positive, the sign code of the result is set to 0x002B; if the input parameter is negative, the sign code of the result is set to 0x002D.

__builtin_packed2zoned

Purpose

Transforms the packed decimal value to zoned decimal format.

This built-in function is valid only when -qarch(-mcpu) is set to utilize the POWER9 technology and the altivec.h file is included.

Prototype

vector unsigned char __builtin_packed2zoned(vector unsigned char, unsigned char);

Usage

The built-in function transforms the packed decimal value to zoned decimal format. The format of the result depends on the value of the second parameter and complies with the following rules:

- If the second parameter is 0, the format of the result is set by the following rules:
  - The leftmost halfword of each digit 0-14 of the result is set to 0x3.
  - The positive sign code is set to 0x3.
  - The negative sign code is set to 0x7.
- If the second parameter is 1, the format of the result is set by the following rules:
  - The leftmost halfword of each digit 0-14 of the result is set to 0xF.
  - The positive sign code is set to 0xC.
  - The negative sign code is set to 0xD.

Notes:

- The second parameter can only be 0 or 1.
- You can determine whether a packed decimal value is positive or negative according to the following rules:
  - Packed decimal values with sign code of 0xA, 0xC, 0xE, or 0xF are interpreted as positive values.
  - Packed decimal values with sign code of 0xB or 0xD are interpreted as negative values.

__builtin_zoned2packed
Purpose

Transforms the zoned decimal value to packed decimal format.

This built-in function is valid only when -qarch(-mcpu) is set to utilize the POWER9 technology and the altivec.h file is included.

Prototype

vector unsigned char __builtin_zoned2packed(vector unsigned char, unsigned char);

Usage

The built-in function transforms the zoned decimal value to packed decimal format. The details of the format of the first parameter depend on the value of the second parameter and comply with the following rules:

- If the second parameter is 0:
  - The value of the first parameter with a sign code of 0x0, 0x1, 0x2, 0x3, 0x8, 0x9, 0xA, or 0xB is interpreted as positive.
  - The value of the first parameter with a sign code of 0x4, 0x5, 0x6, 0x7, 0xC, 0xD, 0xE, or 0xF is interpreted as negative.
- If the second parameter is 1:
  - The value of the first parameter with a sign code of 0xA, 0xC, 0xE, or 0xF is interpreted as positive.

– The value of the first parameter with a sign code of `0xB` or `0xD` is interpreted as negative.

If the first parameter is positive, the sign code for the result is `0xC`; if the first parameter is negative, the sign code for the result is `0xD`.

Note: The second parameter can only be `0` or `1`.

__builtin_bcdcopysign
Purpose

Returns the decimal value of the first parameter combined with the sign code of the second parameter.

This built-in function is valid only when `-qarch`(`-mcpu`) is set to utilize the POWER9 technology and the `altivec.h` file is included.

Prototype

vector unsigned char __builtin_bcdcopysign(vector unsigned char, vector unsigned char);

__builtin_bcdsetsign
Purpose

Set the sign code of the input parameter in packed decimal format.

This built-in function is valid only when `-qarch`(`-mcpu`) is set to utilize the POWER9 technology and the `altivec.h` file is included.

Prototype

vector unsigned char __builtin_bcdsetsign(vector unsigned char, unsigned char);

Usage

Returns the packed decimal value of the first parameter combined with the sign code.

The sign code is set according to the following rules:
- If the packed decimal value of the first parameter is positive, the following rules apply:
  – If the second parameter is `0`, the sign code is set to `0xC`.
  – If the second parameter is `1`, the sign code is set to `0xF`.
- If the packed decimal value of the first parameter is negative, the sign code is set to `0xD`.

Notes:
- The second parameter can only be `0` or `1`.
- You can determine whether a packed decimal value is positive or negative according to the following rules:
  – Packed decimal values with sign code of `0xA`, `0xC`, `0xE`, or `0xF` are interpreted as positive values.

- Packed decimal values with sign code of `0xB` or `0xD` are interpreted as negative values.

__builtin_bcdshift
Purpose

Decimal shift.

This built-in function is valid only when `-qarch(-mcpu)` is set to utilize the POWER9 technology and the `altivec.h` file is included.

Prototype

vector unsigned char __builtin_bcdshift (vector unsigned char a, int b, unsigned char c);

Usage

The built-in function decimal shifts the signed packed decimal value of `a` into the result.

If the value of `b` is positive, `a` is shifted left by `(b<32)?b:31` digits. If the value of `b` is negative, `a` is shifted right by `((-b+1)<32)?(-b+1):31` digits.

The sign code of the result is set according to the following rules:
- If `a` is positive, the sign code is set to `0xD`.
- If `a` is negative, the sign code is set according to the following rules:
  - If `c` equals to `0`, the sign code is set to `0xC`.
  - If `c` equals to `1`, the sign code is set to `0xF`.

Notes:
- You can determine whether a packed decimal value is positive or negative according to the following rules:
  - Packed decimal values with sign code of `0xA`, `0xC`, `0xE`, or `0xF` are interpreted as positive values.
  - Packed decimal values with sign code of `0xB` or `0xD` are interpreted as negative values.
- The value of `c` can only be `0` or `1`.

__builtin_bcdshiftround
Purpose

Decimal shift and round.

This built-in function is valid only when `-qarch(-mcpu)` is set to utilize the POWER9 technology and the `altivec.h` file is included.

Prototype

vector unsigned char __builtin_bcdshiftround (vector unsigned char a, int b, unsigned char c);

Usage

The built-in function decimal shifts and rounds the signed packed decimal value of a into the result.

If the value of b is positive, a is shifted left by b modulo 32 digits. If the value of b is negative, a is shifted right by -b modulo 32 digits. If the value of the last digit shifted out on the right is greater than 5, the result is incremented by 1.

The sign code of the result is set according to the following rules:
- If a is positive, the sign code is set to 0xD.
- If a is negative, the sign code is set according to the following rules:
  - If c equals to 0, the sign code is set to 0xC.
  - If c equals to 1, the sign code is set to 0xF.

The other digits of the result are set to zero.

Note: You can determine whether a packed decimal value is positive or negative according to the following rules:
- Packed decimal values with sign code of 0xA, 0xC, 0xE, or 0xF are interpreted as positive values.
- Packed decimal values with sign code of 0xB or 0xD are interpreted as negative values.

__builtin_bcdtruncate
Purpose

Decimal truncate.

This built-in function is valid only when -qarch(-mcpu) is set to utilize the POWER9 technology and the altivec.h file is included.

Prototype

vector unsigned char __builtin_bcdtruncate(vector unsigned char a, int b, unsigned char c);

Usage

The built-in function copies the rightmost b digits of the signed decimal value of a into the result.

The sign code of the result is set according to the following rules:
- If a is positive, the sign code is set to 0xD.
- If a is negative, the sign code is set according to the following rules:
  - If c equals to 0, the sign code is set to 0xC.
  - If c equals to 1, the sign code is set to 0xF.

The other digits of the result are set to zero.

__builtin_bcdunsignedtruncate
Purpose

Decimal unsigned truncate.

This built-in function is valid only when -qarch(-mcpu) is set to utilize the POWER9 technology and the altivec.h file is included.

Prototype

vector unsigned char __builtin_bcdunsignedtruncate(vector unsigned char a, int b);

Usage

The built-in function copies the rightmost b digits of the unsigned decimal value of a into the result. The other digits of the result are set to zero.

__builtin_bcdunsignedshift
Purpose

Decimal unsigned shift.

This built-in function is valid only when -qarch(-mcpu) is set to utilize the POWER9 technology and the altivec.h file is included.

Prototype

vector unsigned char __builtin_bcdunsignedshift (vector unsigned char a, int b);

Usage

The built-in function decimal shifts the unsigned packed decimal value of a into the result.

If the value of b is positive, a is shifted left by (b<33)?b:32 digits. If the value of b is negative, a is shifted right by ((-b+1)<33)?(-b+1):32 digits.

BCD load and store

The following functions are valid when **-qarch** is set to target POWER7 or higher processors:
- "vec_xl_len_r" on page 503
- "vec_xst_len_r" on page 503

Note: To use the built-in functions, you must include the altivec.h file.

__vec_ldrmb
Purpose

Loads a string of bytes into vector register, right-justified. Sets the leftmost elements (16-*cnt*) to 0.

This built-in function is valid only when you include the altivec.h file.

Prototype

vector unsigned char __vec_ldrmb (char *ptr*, size_t *cnt*);

Parameters

ptr
    Points to a base address.

cnt
> The number of bytes to load. The value of *cnt* must be in the range 1 - 16.

__vec_strmb
Purpose

Stores a right-justified string of bytes.

This built-in function is valid only when you include the `altivec.h` file.

Prototype

> void __vec_strmb (char *ptr*, size_t *cnt*, vector unsigned char *data*);

Parameters

ptr
> Points to a base address.

cnt
> The number of bytes to store. The value of *cnt* must be in the range 1 - 16 and must be a compile-time known constant.

BCD test add and subtract for overflow

The following functions are valid when **-qarch** is set to target POWER8 or higher processors:
- "__bcdadd_ofl"
- "__bcdsub_ofl"
- "__bcd_invalid" on page 350

Note: To use the built-in functions, you must include the `altivec.h` file.

__bcdadd_ofl
Purpose

Returns 1 if the corresponding BCD add operation results in an overflow, or 0 otherwise.

This built-in function is valid only when you include the `altivec.h` file.

Prototype

> long __bcdadd_ofl (vector unsigned char *a*, vector unsigned char *b*);

__bcdsub_ofl
Purpose

Returns 1 if the corresponding BCD subtract operation results in an overflow, or 0 otherwise.

This built-in function is valid only when you include the `altivec.h` file.

Prototype

> long __bcdsub_ofl (vector unsigned char *a*, vector unsigned char *b*);

__bcd_invalid
Purpose

Returns 1 if *a* is an invalid encoding of a BCD value, or 0 otherwise.

This built-in function is valid only when you include the `altivec.h` file.

Prototype

> long __bcd_invalid (vector unsigned char *a*);

Synchronization and atomic built-in functions

Synchronization and atomic built-in functions are grouped into the following categories:
- "Check lock functions"
- "Clear lock functions" on page 351
- "Compare and swap functions" on page 352
- "Fetch functions" on page 353
- "Load functions" on page 354
- "Store functions" on page 355
- "Synchronization functions" on page 356

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 long* *addr*, long long *old_value*, long 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_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

__lqarx, __ldarx, __lwarx, __lharx, __lbarx
Purpose

Load Quadword and Reserve Indexed, Load Doubleword and Reserve Indexed, Load Word and Reserve Indexed, Load Halfword and Reserve Indexed, Load Byte 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

> void __lqarx (volatile long* *addr*, long *dst[2]*);
>
> long __ldarx (volatile long* *addr*);
>
> int __lwarx (volatile int* *addr*);
>
> short __lharx(volatile short* *addr*);
>
> char __lbarx(volatile char* *addr*);

Parameters

addr
> The address of the value to be loaded. Must be aligned on a 4-byte boundary
> for a single word, on an 8-byte boundary for a doubleword, and on a 16-byte
> boundary for a quadword.

dst
> The address to which the value is loaded.

Usage

This function can be used with a subsequent __stqcx (__stdcx, __stwcx, __sthcx,
or __stbcx) 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 have modified
the target memory between the time the load function is executed and the time the
store function completes. This has the same effect on code motion as inserting
__fence built-in functions before and after the load function and can inhibit
compiler optimization of surrounding code (see "__alignx" on page 524 for a
description of the __fence built-in function).

Store functions

__stqcx, __stdcx, __stwcx, __sthcx, __stbcx
Purpose

Store Quadword Conditional Indexed, Store Doubleword Conditional Indexed,
Store Word Conditional Indexed, Store Halfword Conditional Indexed, Store Byte
Conditional Indexed

Stores the value specified by *val* into the memory location specified by *addr*.

Prototype

> int __stqcx(volatile long* *addr*, long *val[2]*);
>
> int __stdcx(volatile long* *addr*, long *val*);
>
> int __stwcx(volatile int* *addr*, int *val*);
>
> int __sthcx(volatile short* *addr*, short *val*);
>
> int __stbcx(volatile char* *addr*, char *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.

val
> The value that 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 __lqarx (__ldarx, __lwarx, __lharx, or __lbarx) 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

Lightweight 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"
- "Prefetch built-in functions" on page 359

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.

__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*);

__dcbtna
Purpose

Data cache block hint no longer accessed

Indicates that the block containing address will not be accessed for a long time; therefore, it must not be kept in the L1 data cache.

Note: Using this function does not necessarily evict the containing block from the data cache.

Prototype

> void __dcbtna (void **addr*);

__dcbtst
Purpose

Data Cache Block Touch for Store

Fetches the block of memory containing the specified address into the data cache.

Prototype

> void __dcbtst (void* *addr*);

__dcbz
Purpose

Data Cache Block set to Zero

Sets a cache line containing the specified address in the data cache to zero (0).

Prototype

> void __dcbz (void* *addr*);

__icbt
Purpose

Instruction cache block touch

Indicates that the program will soon run code in the instruction cache block containing address, and that the block containing address must be loaded into the instruction cache.

Prototype

> void __icbt (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*);

Cryptography built-in functions

Advanced Encryption Standard functions

Advanced Encryption Standard (AES) functions provide support for Federal Information Processing Standards Publication 197 (FIPS-197), which is a specification for encryption and decryption.

__vcipher
Purpose

Performs one round of the AES cipher operation on intermediate state *state_array* using a given *round_key*.

Prototype

> vector unsigned char __vcipher (vector unsigned char *state_array*, vector unsigned char *round_key*);

Parameters

state_array
  The input data chunk to be encrypted or the result of a previous vcipher operation.

round_key
  The 128-bit AES round key value that is used to encrypt.

Result

Returns the resulting intermediate state.

__vcipherlast
Purpose

Performs the final round of the AES cipher operation on intermediate state *state_array* using a given *round_key*.

Prototype

> vector unsigned char __vcipherlast (vector unsigned char *state_array*, vector unsigned char *round_key*);

Parameters

state_array
  The result of a previous vcipher operation.

round_key
  The 128-bit AES round key value that is used to encrypt.

Result

Returns the resulting final state.

__vncipher
Purpose

Performs one round of the AES inverse cipher operation on intermediate state *state_array* using a given *round_key*.

Prototype

> vector unsigned char __vncipher (vector unsigned char *state_array*, vector unsigned char *round_key*);

Parameters

state_array
  The input data chunk to be decrypted or the result of a previous vncipher operation.

round_key
  The 128-bit AES round key value that is used to decrypt.

Result

Returns the resulting intermediate state.

__vncipherlast
Purpose

Performs the final round of the AES inverse cipher operation on intermediate state *state_array* using a given *round_key*.

Prototype

> vector unsigned char __vncipherlast (vector unsigned char *state_array*, vector unsigned char *round_key*);

Parameters

state_array
  The result of a previous vncipher operation.

round_key
  The 128-bit AES round key value that is used to decrypt.

Result

Returns the resulting final state.

__vsbox
Purpose

Performs the SubBytes operation, as defined in FIPS-197, on a *state_array*.

Prototype

> vector unsigned char __vsbox (vector unsigned char *state_array*);

Parameters

state_array
> The input data chunk to be encrypted or the result of a previous vcipher operation.

Result

Returns the result of the operation.

Secure Hash Algorithm functions

Secure Hash Algorithm (SHA) functions provide support for Federal Information Processing Standards Publication 180-3 (FIPS-180-3), Secure Hash Standard. All SHA functions operate on unsigned vector integer types.

__vshasigmad
Purpose

Provides support for Federal Information Processing Standards Publication FIPS-180-3, which is a specification for Secure Hash Standard.

Prototype

> vector unsigned long long __vshasigmad (vector unsigned long long *x*, int *type*, int *fmask*);

Parameters

type
> A compile-time constant in the range 0 - 1. The *type* parameter selects the function type, which can be either lowercase sigma or uppercase sigma.

fmask
> A compile-time constant in the range 0 - 15. The *fmask* parameter selects the function subtype, which can be either sigma-0 or sigma-1.

Result

Let mask be the rightmost 4 bits of fmask.

For each element i (i=0,1) of *x*, element i of the returned value is the following result SHA-512 function:

- The result SHA-512 function is sigma0(x[i]), if type is 0 and bit 2*i of mask is 0.
- The result SHA-512 function is sigma1(x[i]), if type is 0 and bit 2*i of mask is 1.
- The result SHA-512 function is Sigma0(x[i]), if type is non-zero and bit 2*i of mask is 0.
- The result SHA-512 function is Sigma1(x[i]), if type is non-zero and bit 2*i of mask is 1.

__vshasigmaw
Purpose

Provides support for Federal Information Processing Standards Publication FIPS-180-3, which is a specification for Secure Hash Standard.

Prototype

> vector unsigned int __vshasigmaw (vector unsigned int *x*, int *type*, int *fmask*)

Parameters

type
A compile-time constant in the range 0 - 1. The *type* parameter selects the function type, which can be either lowercase sigma or uppercase sigma.

fmask
A compile-time constant in the range 0 - 15. The *fmask* parameter selects the function subtype, which can be either sigma-0 or sigma-1.

Result

Let mask be the rightmost 4 bits of fmask.

For each element i (i=0,1,2,3) of *x*, element i of the returned value is the following result SHA-256 function:

- The result SHA-256 function is sigma0(x[i]), if type is 0 and bit i of mask is 0.
- The result SHA-256 function is sigma1(x[i]), if type is 0 and bit i of mask is 1.
- The result SHA-256 function is Sigma0(x[i]), if type is nonzero and bit i of mask is 0.
- The result SHA-256 function is Sigma1(x[i]), if type is nonzero and bit i of mask is 1.

Miscellaneous functions

__vpermxor
Purpose

Applies a permute and exclusive-OR operation on two byte vectors.

Prototype

> vector unsigned char __vpermxor (vector unsigned char *a*, vector unsigned char *b*, vector unsigned char *mask*);

Result

For each i (0 <= i < 16), let indexA be bits 0 - 3 and indexB be bits 4 - 7 of byte element i of *mask*.

Byte element i of the result is set to the exclusive-OR of byte elements indexA of *a* and indexB of *b*.

Related reference:

"-maltivec (-qaltivec)" on page 129

Related information:

Vector element order toggling

__vpmsumb

Purpose

Performs the exclusive-OR operation on each even-odd pair of the polynomial-multiplication result of corresponding elements.

Prototype

vector unsigned char __vpmsumb (vector unsigned char *a*, vector unsigned char *b*)

Result

For each i (0 <= i < 16), let `prod[i]` be the result of polynomial multiplication of byte elements i of *a* and *b*.

For each i (0 <= i < 8), each halfword element i of the result is set as follows:
- Bit 0 is set to 0.
- Bits 1 - 15 are set to `prod[2*i]` `(xor)` `prod[2*i+1]`.

__vpmsumd

Purpose

Performs the exclusive-OR operation on each even-odd pair of the polynomial-multiplication result of corresponding elements.

Prototype

vector unsigned long long __vpmsumd (vector unsigned long long *a*, vector unsigned long long *b*);

Result

For each i (0 <= i < 2), let `prod[i]` be the result of polynomial multiplication of doubleword elements i of *a* and *b*.

Bit 0 of the result is set to 0.

Bits 1 - 127 of the result are set to `prod[0]` `(xor)` `prod[1]`.

__vpmsumh

Purpose

Performs the exclusive-OR operation on each even-odd pair of the polynomial-multiplication result of corresponding elements.

Prototype

vector unsigned short __vpmsumh (vector unsigned short *a*, vector unsigned short *b*);

Result

For each i (0 <= i < 8), let `prod[i]` be the result of polynomial multiplication of halfword elements i of *a* and *b*.

For each i (0 <= i < 4), each word element i of the result is set as follows:
- Bit 0 is set to 0.
- Bits 1 - 31 are set to prod[2*i] (xor) prod[2*i+1].

__vpmsumw
Purpose

Performs the exclusive-OR operation on each even-odd pair of the polynomial-multiplication result of corresponding elements.

Prototype

vector unsigned int __vpmsumw (vector unsigned int *a*, vector unsigned int *b*);

Result

For each i (0 <= i < 4), let prod[i] be the result of polynomial multiplication of word elements i of *a* and *b*.

For each i (0 <= i < 2), each doubleword element i of the result is set as follows:
- Bit 0 is set to 0.
- Bits 1 - 63 are set to prod[2*i] (xor) prod[2*i+1].

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 Vector Multimedia Extension (VMX) or the Vector Scalar Extension (VSX) built-in functions. This section provides an alphabetical reference to the VMX and the VSX 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. Built-in functions that use or return a **vector unsigned long**

long, **vector signed long long**, **vector bool long long**, or **vector double** type
require an architecture that supports the VSX instruction set extensions.

Function syntax

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 `vector double vec_xld2(int, double*);`
is represented by `d=vec_xld2(a, b)`.

Note:

- This section only describes the IBM specific vector built-in functions and the
  AltiVec built-in functions with IBM extensions. For information about the other
  AltiVec built-in functions, see the AltiVec Application Programming Interface
  specification.
- To use the built-in functions, you must include the `altivec.h` file.

Related reference:

"-maltivec (-qaltivec)" on page 129

vec_abs

Purpose

Returns a vector containing the absolute values of the contents of the given vector.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_abs(a)
```

Result and argument types

The following table describes the types of the returned value and the function
arguments.

Table 43. Types of the returned value and function argument

d	a
vector signed char	vector signed char
vector signed short	vector signed short
vector signed int	vector signed int
vector float	vector float
vector double	vector double

Result value

The value of each element of the result is the absolute value of the corresponding element of a.

vec_absd

Purpose

Returns a vector that contains the absolute difference of the corresponding elements of the given vectors.

This built-in function is valid only when `-qarch(-mcpu)` is set to utilize the POWER9 technology and the `altivec.h` file is included.

Syntax

`d=vec_absd(a, b)`

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 44. Types of the returned value and the function arguments

d	a	b
vector unsigned char	vector unsigned char	vector unsigned char
vector unsigned short	vector unsigned short	vector unsigned short
vector unsigned int	vector unsigned int	vector unsigned int

Result value

The value of each element of the result is the absolute difference of the corresponding elements of a and b using the modulo arithmetic.

vec_abss
Purpose

Returns a vector containing the saturated absolute values of the elements of a given vector.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

`d=vec_abss(a)`

Result and argument types

The following table describes the types of the returned value and the function argument.

Table 45. Types of the returned value and function argument

d	a
vector signed char	vector signed char
vector signed short	vector signed short
vector signed int	vector signed int

Result value

The value of each element of the result is the saturated absolute value of the corresponding element of a.

vec_add

Purpose

Returns a vector containing the sums of each set of corresponding elements of the given vectors.

This function emulates the operation on long long vectors.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_add(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 46. Result and argument types

d	a	b
The same type as argument a	vector signed char	The same type as argument a
	vector unsigned char	
	vector signed short	
	vector unsigned short	
	vector signed int	
	vector unsigned int	
	vector signed long long	
	vector unsigned long long	
	vector float	
	vector double	

Result value

The value of each element of the result is the sum of the corresponding elements of a and b. For integer vectors and unsigned vectors, the arithmetic is modular.

vec_addc

Purpose

Returns a vector containing the carries produced by adding each set of corresponding elements of two given vectors.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

`d=vec_addc(a, b)`

Result and argument types

The type of d, a, and b must be `vector unsigned int`.

Result value

If a carry is produced by adding the corresponding elements of a and b, the corresponding element of the result is 1; otherwise, it is 0.

vec_adds

Purpose

Returns a vector containing the saturated sums of each set of corresponding elements of two given vectors.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

`d=vec_adds(a, b)`

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 47. Types of the returned value and function arguments

d	a	b
vector signed char	vector bool char	vector signed char
	vector signed char	vector bool char
		vector signed char
vector unsigned char	vector bool char	vector unsigned char
	vector unsigned char	vector bool char
		vector unsigned char
vector signed short	vector bool short	vector signed short
	vector signed short	vector bool short
		vector signed short
vector unsigned short	vector bool short	vector unsigned short
	vector unsigned short	vector bool short
		vector unsigned short

Table 47. Types of the returned value and function arguments (continued)

d	a	b
vector signed int	vector bool int	vector signed int
	vector signed int	vector bool int
		vector signed int
vector unsigned int	vector bool int	vector unsigned int
	vector unsigned int	vector bool int
		vector unsigned int

Result value

The value of each element of the result is the saturated sum of the corresponding elements of a and b.

vec_add_u128
Purpose

Adds unsigned quadword values.

The function operates on vectors as 128-bit unsigned integers.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

`d=vec_add_u128(a, b)`

Result and argument types

The type of d, a, and b must be `vector unsigned char`.

Result value

Returns low 128 bits of a + b.

vec_addc_u128
Purpose

Gets the carry bit of the 128-bit addition of two quadword values.

The function operates on vectors as 128-bit unsigned integers.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

`d=vec_addc_u128(a, b)`

Result and argument types

The type of d, a, and b must be `vector unsigned char`.

Result value

Returns the carry out of a + b.

vec_adde_u128
Purpose

Adds unsigned quadword values with carry bit from the previous operation.

The function operates on vectors as 128-bit unsigned integers.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

d=vec_adde_u128(a, b, c)

Result and argument types

The type of d, a, b, and c must be `vector unsigned char`.

Result value

Returns low 128 bits of a + b + (c & 1).

vec_addec_u128
Purpose

Gets the carry bit of the 128-bit addition of two quadword values with carry bit from the previous operation.

The function operates on vectors as 128-bit unsigned integers.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

d=vec_addec_u128(a, b, c)

Result and argument types

The type of d, a, and b must be `vector unsigned char`.

Result value

Returns the carry out of a + b + (c & 1).

vec_all_eq
Purpose

Tests whether all sets of corresponding elements of the given vectors are equal.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

d=vec_all_eq(a, b)

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 48. Result and argument types

d	a	b
int	vector bool char	vector bool char
		vector signed char
		vector unsigned char
	vector signed char	vector bool char
		vector signed char
	vector unsigned char	vector bool char
		vector unsigned char
	vector bool short	vector bool short
		vector signed short
		vector unsigned short
	vector signed short	vector bool short
		vector signed short
	vector unsigned short	vector bool short
		vector unsigned short
	vector bool int	vector bool int
		vector signed int
		vector unsigned int
	vector signed int	vector bool int
		vector signed int
	vector unsigned int	vector bool int
		vector unsigned int
	vector bool long long	vector bool long long
		vector signed long long
		vector unsigned long long
	vector signed long long	vector bool long long
		vector signed long long
	vector unsigned long long	vector bool long long
		vector unsigned long long
	vector float	vector float
	vector double	vector double

Result value

The result is 1 if each element of a is equal to the corresponding element of b. Otherwise, the result is 0.

vec_all_ge

Purpose

Tests whether all elements of the first argument are greater than or equal to the corresponding elements of the second argument.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

`d=vec_all_ge(a, b)`

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 49. Result and argument types

d	a	b
int	vector bool char	vector signed char
		vector unsigned char
	vector signed char	vector bool char
		vector signed char
	vector unsigned char	vector bool char
		vector unsigned char
	vector bool short	vector signed short
		vector unsigned short
	vector signed short	vector bool short
		vector signed short
	vector unsigned short	vector bool short
		vector unsigned short
	vector bool int	vector signed int
		vector unsigned int
	vector signed int	vector bool int
		vector signed int
	vector unsigned int	vector bool int
		vector unsigned int
	vector bool long long	vector signed long long
		vector unsigned long long
	vector signed long long	vector bool long long
		vector signed long long
	vector unsigned long long	vector bool long long
		vector unsigned long long
	vector float	vector float
	vector double	vector double

Result value

The result is 1 if all elements of a are greater than or equal to the corresponding elements of b. Otherwise, the result is 0.

vec_all_gt
Purpose

Tests whether all elements of the first argument are greater than the corresponding elements of the second argument.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_all_gt(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 50. Result and argument types

d	a	b
int	vector bool char	vector signed char
		vector unsigned char
	vector signed char	vector bool char
		vector signed char
	vector unsigned char	vector bool char
		vector unsigned char
	vector bool short	vector signed short
		vector unsigned short
	vector signed short	vector bool short
		vector signed short
	vector unsigned short	vector bool short
		vector unsigned short
	vector bool int	vector signed int
		vector unsigned int
	vector signed int	vector bool int
		vector signed int
	vector unsigned int	vector bool int
		vector unsigned int
	vector bool long long	vector signed long long
		vector unsigned long long
	vector signed long long	vector bool long long
		vector signed long long
	vector unsigned long long	vector bool long long
		vector unsigned long long
	vector float	vector float
	vector double	vector double

Result value

The result is 1 if all elements of a are greater than the corresponding elements of b. Otherwise, the result is 0.

vec_all_in
Purpose

Tests whether each element of a given vector is within a given range.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_all_in(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 51. Types of the returned value and the function arguments

d	a	b
int	vector float	vector float

Result value

The result is 1 if both of the following conditions are satisfied; otherwise, the result is 0.

- All the elements of b have a value greater than or equal to 0.
- All the elements of a have a value less than or equal to the value of the corresponding element of b, and greater than or equal to the negative of the value of the corresponding element of b.

vec_all_le
Purpose

Tests whether all elements of the first argument are less than or equal to the corresponding elements of the second argument.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_all_le(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 52. Result and argument types

d	a	b
int	vector bool char	vector signed char
		vector unsigned char
	vector signed char	vector bool char
		vector signed char
	vector unsigned char	vector bool char
		vector unsigned char
	vector bool short	vector signed short
		vector unsigned short
	vector signed short	vector bool short
		vector signed short
	vector unsigned short	vector bool short
		vector unsigned short
	vector bool int	vector signed int
		vector unsigned int
	vector signed int	vector bool int
		vector signed int
	vector unsigned int	vector bool int
		vector unsigned int
	vector bool long long	vector signed long long
		vector unsigned long long
	vector signed long long	vector bool long long
		vector signed long long
	vector unsigned long long	vector bool long long
		vector unsigned long long
	vector float	vector float
	vector double	vector double

Result value

The result is 1 if all elements of a are less than or equal to the corresponding elements of b. Otherwise, the result is 0.

vec_all_lt
Purpose

Tests whether all elements of the first argument are less than the corresponding elements of the second argument.

This built-in function is valid only when you include the altivec.h file.

Syntax

```
d=vec_all_lt(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 53. Result and argument types

d	a	b
int	vector bool char	vector signed char
		vector unsigned char
	vector signed char	vector bool char
		vector signed char
	vector unsigned char	vector bool char
		vector unsigned char
	vector bool short	vector signed short
		vector unsigned short
	vector signed short	vector bool short
		vector signed short
	vector unsigned short	vector bool short
		vector unsigned short
	vector bool int	vector signed int
		vector unsigned int
	vector signed int	vector bool int
		vector signed int
	vector unsigned int	vector bool int
		vector unsigned int
	vector bool long long	vector signed long long
		vector unsigned long long
	vector signed long long	vector bool long long
		vector signed long long
	vector unsigned long long	vector bool long long
		vector unsigned long long
	vector float	vector float
	vector double	vector double

Result value

The result is 1 if all elements of a are less than the corresponding elements of b. Otherwise, the result is 0.

vec_all_nan
Purpose

Tests whether each element of the given vector is a NaN.

This built-in function is valid only when you include the altivec.h file.

Syntax

```
d=vec_all_nan(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 54. Result and argument types

d	a
int	vector float
	vector double

Result value

The result is 1 if each element of a is a NaN. Otherwise, the result is 0.

vec_all_ne
Purpose

Tests whether all sets of corresponding elements of the given vectors are not equal.

This built-in function is valid only when you include the altivec.h file.

Syntax

```
d=vec_all_ne(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 55. Result and argument types

d	a	b
int	vector bool char	vector signed char
		vector unsigned char
	vector signed char	vector bool char
		vector signed char
	vector unsigned char	vector bool char
		vector unsigned char
	vector bool short	vector signed short
		vector unsigned short
	vector signed short	vector bool short
		vector signed short
	vector unsigned short	vector bool short
		vector unsigned short
	vector bool int	vector signed int
		vector unsigned int
	vector signed int	vector bool int
		vector signed int
	vector unsigned int	vector bool int
		vector unsigned int
	vector bool long long	vector signed long long
		vector unsigned long long
	vector signed long long	vector bool long long
		vector signed long long
	vector unsigned long long	vector bool long long
		vector unsigned long long
	vector float	vector float
	vector double	vector double

Result value

The result is 1 if each element of a is not equal to the corresponding element of b. Otherwise, the result is 0.

vec_all_nge
Purpose

Tests whether each element of the first argument is not greater than or equal to the corresponding element of the second argument.

This built-in function is valid only when you include the altivec.h file.

Syntax

```
d=vec_all_nge(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 56. Result and argument types

d	a	b
int	vector float	vector float
	vector double	vector double

Result value

The result is 1 if each element of a is not greater than or equal to the corresponding element of b. Otherwise, the result is 0.

vec_all_ngt
Purpose

Tests whether each element of the first argument is not greater than the corresponding element of the second argument.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_all_ngt(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 57. Result and argument types

d	a	b
int	vector float	vector float
	vector double	vector double

Result value

The result is 1 if each element of a is not greater than the corresponding element of b. Otherwise, the result is 0.

vec_all_nle
Purpose

Tests whether each element of the first argument is not less than or equal to the corresponding element of the second argument.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_all_nle(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 58. Result and argument types

d	a	b
int	vector float	vector float
	vector double	vector double

Result value

The result is 1 if each element of a is not less than or equal to the corresponding element of b. Otherwise, the result is 0.

vec_all_nlt
Purpose

Tests whether each element of the first argument is not less than the corresponding element of the second argument.

This built-in function is valid only when you include the altivec.h file.

Syntax

```
d=vec_all_nlt(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 59. Result and argument types

d	a	b
int	vector float	vector float
	vector double	vector double

Result value

The result is 1 if each element of a is not less than the corresponding element of b. Otherwise, the result is 0.

vec_all_numeric
Purpose

Tests whether each element of the given vector is numeric (not a NaN).

This built-in function is valid only when you include the altivec.h file.

Syntax

```
d=vec_all_numeric(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 60. Result and argument types

d	a
int	vector float
	vector double

Result value

The result is 1 if each element of `a` is numeric (not a `NaN`). Otherwise, the result is 0.

vec_and

Purpose

Performs a bitwise AND of the given vectors.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

`d=vec_and(a, b)`

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 61. Result and argument types

d	a	b
vector bool char	vector bool char	vector bool char
vector signed char	vector bool char	vector signed char
	vector signed char	vector signed char
		vector bool char
vector unsigned char	vector bool char	vector unsigned char
	vector unsigned char	vector unsigned char
		vector bool char
vector bool short	vector bool short	vector bool short
vector signed short	vector bool short	vector signed short
	vector signed short	vector signed short
		vector bool short
vector unsigned short	vector bool short	vector unsigned short
	vector unsigned short	vector unsigned short
		vector bool short
vector bool int	vector bool int	vector bool int

Table 61. Result and argument types (continued)

d	a	b
vector signed int	vector bool int	vector signed int
	vector signed int	vector signed int
		vector bool int
vector unsigned int	vector bool int	vector unsigned int
	vector unsigned int	vector unsigned int
		vector bool int
vector bool long long	vector bool long long	vector bool long long
vector signed long long	vector bool long long	vector signed long long
	vector signed long long	vector signed long long
		vector bool long long
vector unsigned long long	vector bool long long	vector unsigned long long
	vector unsigned long long	vector unsigned long long
		vector bool long long
vector float	vector bool int	vector float
	vector float	vector bool int
		vector float
vector double	vector bool long long	vector double
	vector double	vector double
		vector bool long long

vec_andc

Purpose

Performs a bitwise AND of the first argument and the bitwise complement of the second argument.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

`d=vec_andc(a, b)`

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 62. Result and argument types

d	a	b
vector bool char	vector bool char	vector bool char
vector signed char	vector bool char	vector signed char
	vector signed char	vector signed char
		vector bool char

Table 62. Result and argument types (continued)

d	a	b
vector unsigned char	vector bool char	vector unsigned char
	vector unsigned char	vector unsigned char
		vector bool char
vector bool short	vector bool short	vector bool short
vector signed short	vector bool short	vector signed short
	vector signed short	vector signed short
		vector bool short
vector unsigned short	vector bool short	vector unsigned short
	vector unsigned short	vector unsigned short
		vector bool short
vector bool int	vector bool int	vector bool int
vector signed int	vector bool int	vector signed int
	vector signed int	vector signed int
		vector bool int
vector unsigned int	vector bool int	vector unsigned int
	vector unsigned int	vector unsigned int
		vector bool int
vector bool long long	vector bool long long	vector bool long long
vector signed long long	vector bool long long	vector signed long long
	vector signed long long	vector signed long long
		vector bool long long
vector unsigned long long	vector bool long long	vector unsigned long long
	vector unsigned long long	vector unsigned long long
		vector bool long long
vector float	vector bool int	vector float
	vector float	vector bool int
		vector float
vector double	vector bool long long	vector double
	vector double	vector bool long long
		vector double

Result value

The result is the bitwise AND of `a` with the bitwise complement of `b`.

vec_any_eq
Purpose

Tests whether any set of corresponding elements of the given vectors are equal.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_any_eq(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 63. Result and argument types

d	a	b
int	vector bool char	vector bool char
		vector signed char
		vector unsigned char
	vector signed char	vector bool char
		vector signed char
	vector unsigned char	vector bool char
		vector unsigned char
	vector bool short	vector bool short
		vector signed short
		vector unsigned short
	vector signed short	vector bool short
		vector signed short
	vector unsigned short	vector bool short
		vector unsigned short
	vector bool int	vector bool int
		vector signed int
		vector unsigned int
	vector signed int	vector bool int
		vector signed int
	vector unsigned int	vector bool int
		vector unsigned int
	vector bool long long	vector bool long long
		vector signed long long
		vector unsigned long long
	vector signed long long	vector bool long long
		vector signed long long
	vector unsigned long long	vector bool long long
		vector unsigned long long
	vector float	vector float
	vector double	vector double

Result value

The result is 1 if any element of a is equal to the corresponding element of b. Otherwise, the result is 0.

vec_any_ge
Purpose

Tests whether any element of the first argument is greater than or equal to the corresponding element of the second argument.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

`d=vec_any_ge(a, b)`

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 64. Result and argument types

d	a	b
int	vector bool char	vector signed char
		vector unsigned char
	vector signed char	vector bool char
		vector signed char
	vector unsigned char	vector bool char
		vector unsigned char
	vector bool short	vector signed short
		vector unsigned short
	vector signed short	vector signed short
		vector bool short
	vector unsigned short	vector bool short
		vector unsigned short
	vector bool int	vector signed int
		vector unsigned int
	vector signed int	vector bool int
		vector signed int
	vector unsigned int	vector bool int
		vector unsigned int
	vector bool long long	vector signed long long
		vector unsigned long long
	vector signed long long	vector bool long long
		vector signed long long
	vector unsigned long long	vector bool long long
		vector unsigned long long
	vector float	vector float
	vector double	vector double

Result value

The result is 1 if any element of a is greater than or equal to the corresponding element of b. Otherwise, the result is 0.

vec_any_gt
Purpose

Tests whether any element of the first argument is greater than the corresponding element of the second argument.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

`d=vec_any_gt(a, b)`

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 65. Result and argument types

d	a	b
int	vector bool char	vector signed char
		vector unsigned char
	vector signed char	vector bool char
		vector signed char
	vector unsigned char	vector bool char
		vector unsigned char
	vector bool short	vector signed short
		vector unsigned short
	vector signed short	vector signed short
		vector bool short
	vector unsigned short	vector bool short
		vector unsigned short
	vector bool int	vector signed int
		vector unsigned int
	vector signed int	vector bool int
		vector signed int
	vector unsigned int	vector bool int
		vector unsigned int
	vector bool long long	vector signed long long
		vector unsigned long long
	vector signed long long	vector bool long long
		vector signed long long
	vector unsigned long long	vector bool long long
		vector unsigned long long
	vector float	vector float
	vector double	vector double

Result value

The result is 1 if any element of a is greater than the corresponding element of b. Otherwise, the result is 0.

vec_any_le
Purpose

Tests whether any element of the first argument is less than or equal to the corresponding element of the second argument.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_any_le(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 66. Result and argument types

d	a	b
int	vector bool char	vector signed char
		vector unsigned char
	vector signed char	vector bool char
		vector signed char
	vector unsigned char	vector bool char
		vector unsigned char
	vector bool short	vector signed short
		vector unsigned short
	vector signed short	vector signed short
		vector bool short
	vector unsigned short	vector bool short
		vector unsigned short
	vector bool int	vector signed int
		vector unsigned int
	vector signed int	vector bool int
		vector signed int
	vector unsigned int	vector bool int
		vector unsigned int
	vector bool long long	vector signed long long
		vector unsigned long long
	vector signed long long	vector bool long long
		vector signed long long
	vector unsigned long long	vector bool long long
		vector unsigned long long
	vector float	vector float
	vector double	vector double

Result value

The result is 1 if any element of a is less than or equal to the corresponding element of b. Otherwise, the result is 0.

vec_any_lt
Purpose

Tests whether any element of the first argument is less than the corresponding element of the second argument.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_any_lt(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 67. Result and argument types

d	a	b
int	vector bool char	vector signed char
		vector unsigned char
	vector signed char	vector bool char
		vector signed char
	vector unsigned char	vector bool char
		vector unsigned char
	vector bool short	vector signed short
		vector unsigned short
	vector signed short	vector signed short
		vector bool short
	vector unsigned short	vector bool short
		vector unsigned short
	vector bool int	vector signed int
		vector unsigned int
	vector signed int	vector bool int
		vector signed int
	vector unsigned int	vector bool int
		vector unsigned int
	vector bool long long	vector signed long long
		vector unsigned long long
	vector signed long long	vector bool long long
		vector signed long long
	vector unsigned long long	vector bool long long
		vector unsigned long long
	vector float	vector float
	vector double	vector double

Result value

The result is 1 if any element of a is less than the corresponding element of b. Otherwise, the result is 0.

vec_any_nan

Purpose

Tests whether any element of the given vector is a NaN.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

`d=vec_any_nan(a)`

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 68. Result and argument types

d	a
int	vector float
	vector double

Result value

The result is 1 if any element of a is a NaN. Otherwise, the result is 0.

vec_any_ne

Purpose

Tests whether any set of corresponding elements of the given vectors are not equal.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

`d=vec_any_ne(a, b)`

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 69. Result and argument types

d	a	b
int	vector bool char	vector bool char
		vector signed char
		vector unsigned char
	vector signed char	vector bool char
		vector signed char
	vector unsigned char	vector bool char
		vector unsigned char
	vector bool short	vector bool short
		vector signed short
		vector unsigned short
	vector signed short	vector bool short
		vector signed short
	vector unsigned short	vector bool short
		vector unsigned short
	vector bool int	vector bool int
		vector signed int
		vector unsigned int
	vector signed int	vector bool int
		vector signed int
	vector unsigned int	vector bool int
		vector unsigned int
	vector bool long long	vector bool long long
		vector signed long long
		vector unsigned long long
	vector signed long long	vector bool long long
		vector signed long long
	vector unsigned long long	vector bool long long
		vector unsigned long long
	vector float	vector float
	vector double	vector double

Result value

The result is 1 if any element of a is not equal to the corresponding element of b. Otherwise, the result is 0.

vec_any_nge
Purpose

Tests whether any element of the first argument is not greater than or equal to the corresponding element of the second argument.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

d=vec_any_nge(a, b)

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 70. Result and argument types

d	a	b
int	vector float	vector float
	vector double	vector double

Result value

The result is 1 if any element of `a` is not greater than or equal to the corresponding element of b. Otherwise, the result is 0.

vec_any_ngt
Purpose

Tests whether any element of the first argument is not greater than the corresponding element of the second argument.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

d=vec_any_ngt(a, b)

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 71. Result and argument types

d	a	b
int	vector float	vector float
	vector double	vector double

Result value

The result is 1 if any element of `a` is not greater than the corresponding element of b. Otherwise, the result is 0.

vec_any_nle
Purpose

Tests whether any element of the first argument is not less than or equal to the corresponding element of the second argument.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_any_nle(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 72. Result and argument types

d	a	b
int	vector float	vector float
	vector double	vector double

Result value

The result is 1 if any element of `a` is not less than or equal to the corresponding element of `b`. Otherwise, the result is 0.

vec_any_nlt
Purpose

Tests whether any element of the first argument is not less than the corresponding element of the second argument.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_any_nlt(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 73. Result and argument types

d	a	b
int	vector float	vector float
	vector double	vector double

Result value

The result is 1 if any element of `a` is not less than the corresponding element of `b`. Otherwise, the result is 0.

vec_any_numeric
Purpose

Tests whether any element of the given vector is numeric (not a `NaN`).

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_any_numeric(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 74. Result and argument types

d	a
int	vector float
	vector double

Result value

The result is 1 if any element of `a` is numeric (not a NaN). Otherwise, the result is 0.

vec_any_out
Purpose

Tests whether the value of any element of a given vector is outside of a given range.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_any_out(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 75. Types of the returned value and the function arguments

d	a	b
int	vector float	vector float

Result value

The result is 1 if both of the following conditions are satisfied; otherwise, the result is 0.

- All the elements of b have a value greater than or equal to 0.
- The absolute value of any element of a is greater than the value of the corresponding element of b or less than the negative of the value of the corresponding element of b.

vec_avg
Purpose

Returns a vector containing the rounded average of each set of corresponding elements of two given vectors.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_avg(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 76. Types of the returned value and function arguments

d	a	b
The same type as argument a	vector signed char	The same type as argument a
	vector unsigned char	
	vector signed short	
	vector unsigned short	
	vector signed int	
	vector unsigned int	

Result value

The value of each element of the result is the rounded average of the values of the corresponding elements of `a` and `b`.

vec_bperm

Purpose

Gathers up to 16 1-bit values from a quadword or from each doubleword element in the specified order, and places them in the specified order either in the rightmost 16 bits of the leftmost doubleword of the result vector register or in the rightmost 8 bits of each doubleword of the result vector register according to the element types, with the rest of the result set to 0.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_bperm(a, b)
```

Result and argument types

The following table describes the types of the result and the function arguments.

Table 77. Types of the returned value and function arguemnts

d	a	b
vector unsigned char	vector unsigned char[1]	vector unsigned char[1]
vector unsigned long long	vector unsigned long long[2]	vector unsigned char[2]

Note:

1. This combination of data types of the arguments of the built-in function is valid only when `-qarch`(`-mcpu`) is set to utilize POWER8 or POWER9 technologies.

2. This combination of data types of the arguments of the built-in function is valid only when -qarch(-mcpu) is set to utilize POWER9 technology.

Result value

- When the data type of a is vector unsigned char, which is valid only when -qarch is set to target POWER8 or POWER9:

  Suppose $i(0<=i<16)$ and j. Let i denote the element index of b, and let j denote the byte value of element i of b:
  - If $j>=128$, bit *48+i* of doubleword 0 is set to 0.
  - If $j<128$, bit *48+i* of the result is set to the value of bit j of a.
  - All other bits are set to 0.
- When the data type of a is vector unsigned long long, which is valid only when -qarch is set to target POWER9:

  Suppose $i(0<=i<2)$, $j(0<=j<8)$ and k. Let i denote the doubleword element index of a; let j denote the element index of b; and let k denote the byte value of element j of b:
  - If $k>=64$, bit *56+j* of element i is set to 0.
  - If $k<64$, bit *56+j* of element i is set to the value of bit k of element i of a.
  - All other bits are set to 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.

Note: vec_ceil is another name for vec_roundp. For details, see "vec_roundp" on page 467.

This built-in function is valid only when you include the altivec.h file.

vec_cipher_be

Purpose

Performs one round of the AES cipher operation, as defined in *Federal Information Processing Standards Publication 197 (FIPS-197)*, on an intermediate state a by using a given round key b.

This built-in function is valid only when you include the altivec.h file.

Syntax

```
d=vec_cipher_be(a, b)
```

Result and argument types

The type of d, a, and b must be vector unsigned char.

Result value

Returns the resulting intermediate state.

vec_cipherlast_be

Purpose

Performs the final round of the AES cipher operation, as defined in *Federal Information Processing Standards Publication 197 (FIPS-197)*, on an intermediate state a by using a given round key b.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_cipherlast_be(a, b)
```

Result and argument types

The type of d, a, and b must be `vector unsigned char`.

Result value

Returns the resulting final state.

vec_cmpb

Purpose

Performs a bounds comparison of each set of corresponding elements of the given vectors.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_cmpb(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 78. Types of the returned value and function arguments

d	a	b
vector signed int	vector float	vector float

Result value

Each element of the result has the value 0 if both of the following conditions are satisfied:
- All the elements of b have a value greater than or equal to 0.
- All the elements of a have a value less than or equal to the value of the corresponding element of b and greater than or equal to the negative of the value of the corresponding element of b.

Otherwise, the result is determined as follows:
- If an element of b is greater than or equal to zero, the following rules comply:

- If the absolute value of the corresponding element of a is equal to the value of the corresponding element of b, the value of the corresponding element of the result is 0.
- If the absolute value of the corresponding element of a is greater than the value of the corresponding element of b, the value of the corresponding element of the result is negative.
- If the absolute value of the corresponding element of a is less than the value of the corresponding element of b, the value of the corresponding element of the result is positive.
- If an element of b is less than zero, the value of the element of the result is positive if the value of the corresponding element of a is less than or equal to the value of the element of b, and negative otherwise.

vec_cmpeq

Purpose

Returns a vector containing the results of comparing each set of corresponding elements of the given vectors for equality.

This function emulates the operation on long long vectors.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

`d=vec_cmpeq(a, b)`

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 79. Result and argument types

d	a	b
vector bool char	vector bool char	The same type as argument a
	vector signed char	
	vector unsigned char	
vector bool short	vector bool short	
	vector signed short	
	vector unsigned short	
vector bool int	vector bool int	
	vector signed int	
	vector unsigned int	
	vector float	
vector bool long long	vector bool long long	
	vector signed long long	
	vector unsigned long long	
	vector double	

Result value

For each element of the result, the value of each bit is 1 if the corresponding elements of a and b are equal. Otherwise, the value of each bit is 0.

vec_cmpge

Purpose

Returns a vector containing the results of a greater-than-or-equal-to comparison between each set of corresponding elements of the given vectors.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

`d=vec_cmpge(a, b)`

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 80. Result and argument types

d	a	b
vector bool char	vector signed char	The same type as argument a
	vector unsigned char	
vector bool short	vector signed short	
	vector unsigned short	
vector bool int	vector signed int	
	vector unsigned int	
	vector float	
vector bool long long	vector signed long long	
	vector unsigned long long	
	vector double	

Result value

For each element of the result, the value of each bit is 1 if the value of the corresponding element of a is greater than or equal to the value of the corresponding element of b. Otherwise, the value of each bit is 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.

This function emulates the operation on long long vectors.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_cmpgt(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 81. Result and argument types

d	a	b
vector bool char	vector signed char	vector signed char
	vector unsigned char	vector unsigned char
vector bool short	vector signed short	vector signed short
	vector unsigned short	vector unsigned short
vector bool int	vector signed int	vector signed int
	vector unsigned int	vector unsigned int
	vector float	vector float
vector bool long long	vector signed long long	vector signed long long
	vector unsigned long long	vector unsigned long long
	vector double	vector double

Result value

For each element of the result, the value of each bit is 1 if the value of the corresponding element of a is greater than the value of the corresponding element of b. Otherwise, the value of each bit is 0.

vec_cmple
Purpose

Returns a vector containing the results of a less-than-or-equal-to comparison between each set of corresponding elements of the given vectors.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_cmple(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 82. Result and argument types

d	a	b
vector bool char	vector signed char	vector signed char
	vector unsigned char	vector unsigned char
vector bool short	vector signed short	vector signed short
	vector unsigned short	vector unsigned short

Table 82. Result and argument types (continued)

d	a	b
vector bool int	vector signed int	vector signed int
	vector unsigned int	vector unsigned int
	vector float	vector float
vector bool long long	vector signed long long	vector signed long long
	vector unsigned long long	vector unsigned long long
	vector double	vector double

Result value

For each element of the result, the value of each bit is 1 if the value of the corresponding element of a is less than or equal to the value of the corresponding element of b. Otherwise, the value of each bit is 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.

This operation emulates the operation on long long vectors.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_cmplt(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 83. Result and argument types

d	a	b
vector bool char	vector signed char	vector signed char
	vector unsigned char	vector unsigned char
vector bool short	vector signed short	vector signed short
	vector unsigned short	vector unsigned short
vector bool int	vector signed int	vector signed int
	vector unsigned int	vector unsigned int
	vector float	vector float
vector bool long long	vector signed long long	vector signed long long
	vector unsigned long long	vector unsigned long long
	vector double	vector double

Result value

For each element of the result, the value of each bit is 1 if the value of the
corresponding element of a is less than the value of the corresponding element of
b. Otherwise, the value of each bit is 0.

vec_cmpne

Purpose

Returns a vector containing the results of comparing each set of the corresponding
elements of the given vectors for inequality.

This built-in function is valid only when -qarch(-mcpu) is set to utilize the
POWER9 technology and the altivec.h file is included.

Syntax

```
d=vec_cmpne(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function
arguments.

Table 84. Types of the returned value and function arguments

d	a	b
vector bool char	vector signed char	the same type as a
	vector unsigned char	
	vector bool char	
vector bool int	vector signed int	
	vector unsigned int	
	vector bool int	
	vector float	
vector bool short	vector signed short	
	vector unsigned short	
	vector bool short	
vector bool long long	vector signed long long	
	vector unsigned long long	
	vector bool long long	
	vector double	

Result value

For each element of the result, the value of each bit is 1 if the corresponding
elements of a and b are not equal; otherwise, the value is 0.

vec_cmpnez

Purpose

Returns a vector that contains the results of comparing each set of the corresponding elements of the given vectors for inequality, or the results of testing the corresponding element of given vectors for the value of zero.

This built-in function is valid only when -qarch(-mcpu) is set to utilize the POWER9 technology and the altivec.h file is included.

Syntax

d=vec_cmpeq(a, b)

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 85. Types of the returned value and function arguments

d	a	b
vector bool char	vector signed char	the same type as a
	vector unsigned char	
vector bool short	vector signed short	
	vector unsigned short	
vector bool int	vector signed int	
	vector unsigned int	

Result value

For each element of the result, the value of each bit is 1 in one of the following cases; otherwise, the value of each bit is 0.
- the corresponding elements of a and b are not equal.
- the corresponding element of a or b is 0.

vec_cntlz

Purpose

Counts the most significant zero bits of each element of the given vector.

This built-in function is valid only when -qarch(-mcpu) is set to utilize the POWER9 technology and the altivec.h file is included.

Syntax

d=vec_cntlz(a)

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 86. Result and argument types

d	a
vector signed char	vector signed char
vector unsigned char	vector unsigned char
vector signed short	vector signed short
vector unsigned short	vector unsigned short
vector signed int	vector signed int
vector unsigned int	vector unsigned int
vector signed long long	vector signed long long
vector unsigned long long	vector unsigned long long

Result value

The value of each element of the result is set to the number of leading zeros of the corresponding element of a.

vec_cntlz_lsbb
Purpose

Counts the leading byte elements of the given vector that have a least significant bit of 0.

This built-in function is valid only when -qarch(-mcpu) is set to utilize the POWER9 technology and the altivec.h file is included.

Syntax

```
d=vec_cntlz(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 87. Result and argument types

d	a
signed int	vector signed char
	vector unsigned char

Result value

The result is set to the number of leading byte elements of a that have a least significant bit of 0.

vec_cnttz
Purpose

Counts the least significant zero bits of each element of the given vector.

This built-in function is valid only when -qarch(-mcpu) is set to utilize the
POWER9 technology and the altivec.h file is included.

Syntax

d=vec_cnttz(a)

Result and argument types

The following table describes the types of the returned value and the function
argument.

Table 88. Result and argument types

d	a
vector signed char	vector signed char
vector unsigned char	vector unsigned char
vector signed short	vector signed short
vector unsigned short	vector unsigned short
vector signed int	vector signed int
vector unsigned int	vector unsigned int
vector signed long long	vector signed long long
vector unsigned long long	vector unsigned long long

Result value

The value of each element of the result is set to the number of trailing zeros of the
corresponding element of a.

vec_cnttz_lsbb

Purpose

Counts the trailing byte elements of the given vector that have a least significant
bit of 0.

This built-in function is valid only when -qarch(-mcpu) is set to utilize the
POWER9 technology and the altivec.h file is included.

Syntax

d=vec_cnttz_lsbb(a)

Result and argument types

The following table describes the types of the returned value and the function
arguments.

Table 89. Result and argument types

d	a
vector signed int	vector signed char
	vector unsigned char

Result value

The result is set to the number of trailing byte elements of a that have a least significant bit of 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.

This built-in function is valid only when you include the altivec.h file.

Syntax

d=vec_cpsgn(a, b)

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 90. Result and argument types

d	a	b
vector float	vector float	vector float
vector double	vector double	vector double

vec_ctd
Purpose

Converts each element in a from an integer number to a floating-point number with single precision, and divides the result by 2 to the power of b.

This built-in function is valid only when you include the altivec.h file.

Syntax

d=vec_ctd(a, b)

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 91. Result and argument types

d	a	b
vector double	vector signed int	0-31
	vector unsigned int	
	vector signed long long	
	vector unsigned long long	

vec_ctf

Purpose

Converts a vector of fixed-point numbers into a vector of floating-point numbers.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

`d=vec_ctf(a, b)`

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 92. Result and argument types

d	a	b
vector float	vector signed int	0-31
	vector unsigned int	
	vector signed long long	
	vector unsigned long long	

Result value

The value of each element of the result is the closest floating-point estimate of the value of the corresponding element of a divided by 2 to the power of b.

Note: The second and fourth elements of the result vector are undefined when a is a signed long long or an unsigned long long vector.

vec_cts

Purpose

Converts a vector of floating-point numbers into a vector of signed fixed-point numbers.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

`d=vec_cts(a, b)`

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 93. Result and argument types

d	a	b
vector signed int	vector float	0-31
	vector double	

Result value

The value of each element of the result is the saturated value obtained by multiplying the corresponding element of a by 2 to the power of b.

vec_ctsl
Purpose

Multiplies each element in a by 2 to the power of b and converts the result into an integer.

Note: This function does not use elements 1 and 3 of a when a is a double vector.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_ctsl(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 94. Result and argument types

d	a	b
vector signed long long	vector float	0-31
	vector double	

vec_ctu
Purpose

Converts a vector of floating-point numbers into a vector of unsigned fixed-point numbers.

Note: Elements 1 and 3 of the result vector are undefined when a is a double vector.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_ctu(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 95. Result and argument types

d	a	b
vector unsigned int	vector float	0-31
	vector double	

Result value

The value of each element of the result is the saturated value obtained by multiplying the corresponding element of a by 2 to the power of b.

vec_ctul

Purpose

Multiplies each element in a by 2 to the power of b and converts the result into an unsigned type.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_ctul(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 96. Result and argument types

d	a	b
vector unsigned long long	vector float	0-31
	vector double	

Result value

This function does not use elements 1 and 3 of a when a is a float vector.

vec_cvf

Purpose

Converts a single-precision floating-point vector to a double-precision floating-point vector or converts a double-precision floating-point vector to a single-precision floating-point vector.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_cvf(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 97. Result and argument types

d	a
vector float	vector double
vector double	vector float

Result value

When this function converts from vector float to vector double, it converts the types of elements 0 and 2 in the vector.

When this function converts from vector double to vector float, the types of element 1 and 3 in the result vector are undefined.

vec_div

Purpose

Divides the elements in vector a by the corresponding elements in vector b and then assigns the result to corresponding elements in the result vector.

This function emulates the operation on integer vectors.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_div(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 98. Result and argument types

d	a	b
The same type as argument a	vector signed char	The same type as argument a
	vector unsigned char	
	vector signed short	
	vector unsigned short	
	vector signed int	
	vector unsigned int	
	vector signed long long	
	vector unsigned long long	
	vector float	
	vector double	

vec_dss

Purpose

Stops the data stream read specified by a.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
vec_dss(a)
```

Result and argument types

a must be a 2-bit unsigned literal. This function does not return any value.

vec_dssall
Purpose

Stops all data stream reads.

This built-in function is valid only when you include the `altivec.h` file.

Syntax
```
vec_dssall()
```

vec_dst
Purpose

Initiates the data read of a line into cache in a state most efficient for reading.

The data stream specified by c is read beginning at the address specified by a using the control word specified by b. After you use this built-in function, the specified data stream is relatively persistent.

This built-in function is valid only when you include the `altivec.h` file.

Syntax
```
vec_dst(a, b, c)
```

Argument types

This function does not return any value. The following table describes the types of the function arguments.

Table 99. Types of the function arguments

a	b	c[1]
const signed char *	any integral type	unsigned int
const signed short *		
const signed int *		
const float *		

Note:
1. c must be an unsigned literal with a value in the range 0 - 3 inclusive.

vec_dstst
Purpose

Initiates the data read of a line into cache in a state most efficient for writing.

The data stream specified by c is read beginning at the address specified by a using the control word specified by b. Use of this built-in function indicates that the specified data stream is relatively persistent in nature.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
vec_dstst(a, b, c)
```

Argument types

This function does not return any value. The following table describes the types of the function arguments.

Table 100. Types of the function arguments

a	b	c[1]
const signed char *	any integral type	unsigned int
const signed short *		
const signed int *		
const float *		

Note:

1. `c` must be an unsigned literal with a value in the range 0 - 3 inclusive.

vec_dststt
Purpose

Initiates the data read of a line into cache in a state most efficient for writing.

The data stream specified by `c` is read beginning at the address specified by `a` using the control word specified by `b`. Use of this built-in function indicates that the specified data stream is relatively transient in nature.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
vec_dststt(a, b, c)
```

Argument types

This function does not return a value. The following table describes the types of the function arguments.

Table 101. Types of the function arguments

a	b	c[1]
const signed char *	any integral type	unsigned int
const signed short *		
const signed int *		
const float *		

Note:

1. `c` must be an unsigned literal with a value in the range 0 - 3 inclusive.

vec_dstt

Purpose

Initiates the data read of a line into cache in a state most efficient for reading.

The data stream specified by `c` is read beginning at the address specified by `a` using the control word specified by `b`. Use of this built-in function indicates that the specified data stream is relatively transient in nature.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

`vec_dstt(a, b, c)`

Argument types

This function does not return a value. The following table describes the types of the function arguments.

Table 102. Types of the function arguments

a	b	c[1]
const signed char *	any integral type	unsigned int
const signed short *		
const signed int *		
const float *		

Note:
1. `c` must be an unsigned literal with a value in the range 0 - 3 inclusive.

vec_eqv

Purpose

Performs a bitwise equivalence operation on the input vectors.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

`d=vec_eqv(a, b)`

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 103. Types of the returned value and function arguments

d	a	b
vector signed char	vector signed char	vector signed char
		vector bool char
vector unsigned char	vector unsigned char	vector unsigned char
		vector bool char

Table 103. Types of the returned value and function arguments (continued)

d	a	b
vector signed char	vector bool char	vector signed char
vector unsigned char		vector unsigned char
vector bool char		vector bool char
vector signed short	vector signed short	vector signed short
		vector bool short
vector unsigned short	vector unsigned short	vector unsigned short
		vector bool short
vector signed short	vector bool short	vector signed short
vector unsigned short		vector unsigned short
vector bool short		vector bool short
vector signed int	vector signed int	vector signed int
		vector bool int
vector unsigned int	vector unsigned int	vector unsigned int
		vector bool int
vector signed int	vector bool int	vector signed int
vector unsigned int		vector unsigned int
vector bool int		vector bool int
vector signed long long	vector signed long long	vector signed long long
		vector bool long long
vector unsigned long long	vector unsigned long long	vector unsigned long long
		vector bool long long
vector signed long long	vector bool long long	vector signed long long
vector unsigned long long		vector unsigned long long
vector bool long long		vector bool long long
vector float	vector float	vector bool int
		vector float
	vector bool int	vector float
vector double	vector double	vector double
		vector bool long long
	vector bool long long	vector double

Result value

Each bit of the result is set to the result of the bitwise operation (a == b) of the
corresponding bits of a and b. For 0 <= i < 128, bit i of the result is set to 1 only if
bit i of a is equal to bit i of b.

vec_expte
Purpose

Returns a vector containing estimates of 2 raised to the values of the corresponding
elements of a given vector.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_expte(a)
```

Result and argument types

The type of d and a must be `vector float`.

Result value

Each element of the result contains the estimated value of 2 raised to the value of the corresponding element of a.

vec_extsbd

Purpose

Sign-extends the rightmost byte of each doubleword element of the given vector.

This built-in function is valid only when `-qarch(-mcpu)` is set to utilize the POWER9 technology and the `altivec.h` file is included.

Syntax

```
d=vec_extsbd(a)
```

Result and argument types

The following table describes the types of the returned value and the function argument.

Table 104. Result and argument types

d	a
vector signed long long	vector signed char
	vector unsigned char

Result value

For each doubleword element of a, the rightmost byte is sign-extended and placed into the corresponding doubleword element of d.

vec_extsbw

Purpose

Sign-extends the rightmost byte of each word element of the given vector.

This built-in function is valid only when `-qarch(-mcpu)` is set to utilize the POWER9 technology and the `altivec.h` file is included.

Syntax

```
d=vec_extsbw(a)
```

Result and argument types

The following table describes the types of the returned value and the function argument.

Table 105. Result and argument types

d	a
vector signed int	vector signed char
	vector unsigned char

Result value

For each word element of a, the rightmost byte is sign-extended and placed into the corresponding word element of d.

vec_extshd

Purpose

Sign-extends the rightmost halfword of each doubleword element of the given vector.

This built-in function is valid only when -qarch(-mcpu) is set to utilize the POWER9 technology and the altivec.h file is included.

Syntax

```
d=vec_extshd(a)
```

Result and argument types

The following table describes the types of the returned value and the function argument.

Table 106. Result and argument types

d	a
vector signed long long	vector signed short
	vector unsigned short

Result value

For each doubleword element of a, the rightmost halfword is sign-extended and placed into the corresponding doubleword element of d.

vec_extshw

Purpose

Sign-extends the rightmost halfword of each word element of the given vector.

This built-in function is valid only when -qarch(-mcpu) is set to utilize the POWER9 technology and the altivec.h file is included.

Syntax

```
d=vec_extshw(a)
```

Result and argument types

The following table describes the types of the returned value and the function argument.

Table 107. Result and argument types

d	a
vector signed int	vector signed short
	vector unsigned short

Result value

For each word element of a, the rightmost halfword is sign-extended and placed into the corresponding word element of d.

vec_extswd

Purpose

Sign-extends the rightmost word of each doubleword element of the given vector.

This built-in function is valid only when -qarch(-mcpu) is set to utilize the POWER9 technology and the altivec.h file is included.

Syntax

```
d=vec_extswd(a)
```

Result and argument types

The following table describes the types of the returned value and the function argument.

Table 108. Result and argument types

d	a
vector signed long long	vector signed int
	vector unsigned int

Result value

For each doubleword element of a, the rightmost word is sign-extended and placed into the corresponding doubleword element of d.

vec_extract

Purpose

Returns the value of element b from the vector a.

This built-in function is valid only when you include the altivec.h file.

Syntax

```
d=vec_extract(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 109. Result and argument types

d	a	b
signed char	vector signed char	signed int
unsigned char	vector unsigned char	
	vector bool char	
signed short	vector signed short	
unsigned short	vector unsigned short	
	vector bool short	
signed int	vector signed int	
unsigned int	vector unsigned int	
	vector bool int	
signed long long	vector signed long long	
unsigned long long	vector unsigned long long	
	vector bool long long	
float	vector float	
double	vector double	

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 a to determine the element position.

vec_extract_exp

Purpose

Returns a vector that contains the exponent of the given vector.

This built-in function is valid only when -qarch(-mcpu) is set to utilize the POWER9 technology and the altivec.h file is included.

Syntax

```
d=vec_extract_exp(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 110. Types of the returned value and function argument

d	a
vector unsigned int	vector float
vector unsigned long long	vector double

Result value

Each element of the result is extracted from the exponent field of the corresponding element of a.

Note: The extracted exponent of a is treated as a biased exponent in accordance with the format specified by *IEEE Standard for Binary Floating-Point Arithmetic, IEEE Std. 754*, without further processing.

vec_extract_sig

Purpose

Returns a vector that contains the significand of the given vector.

This built-in function is valid only when `-qarch(-mcpu)` is set to utilize the POWER9 technology and the `altivec.h` file is included.

Syntax

`d=vec_extract_sig(a)`

Result and argument types

The following table describes the types of the returned value and the function argument.

Table 111. Types of the returned value and function argument

d	a
vector unsigned int	vector float
vector unsigned long long	vector double

Result value

Each element of the result is extracted from the significand field of the corresponding element of a.

Note:
- The extracted significand of a is in accordance with the format specified by *IEEE Standard for Binary Floating-Point Arithmetic, IEEE Std. 754*, without further processing.
- The value of the implicit leading digit that is included in the result is not encoded in the *IEEE Standard for Binary Floating-Point Arithmetic, IEEE Std. 754*, but implied by the exponent.

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.

Note: `vec_floor` is another name for `vec_roundm`. For details, see "vec_roundm" on page 467.

This built-in function is valid only when you include the `altivec.h` file.

vec_first_match_index

Purpose

Compares each set of the corresponding elements of the given vectors and returns the first position of equality.

This built-in function is valid only when `-qarch(-mcpu)` is set to utilize the POWER9 technology and the `altivec.h` file is included.

Syntax

`d=vec_first_match_index(a, b)`

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 112. Result and argument types

d	a	b
unsigned int	vector signed char	vector signed char
unsigned int	vector unsigned char	vector unsigned char
unsigned int	vector signed short	vector signed short
unsigned int	vector unsigned short	vector unsigned short
unsigned int	vector signed int	vector signed int
unsigned int	vector unsigned int	vector unsigned int

Result value

Returns the byte index of the position of the first equal element between a and b. If all the corresponding elements are unequal between a and b, returns the number of the elements of either a or b.

vec_first_match_or_eos_index

Purpose

Compares each set of the corresponding elements of the given vectors and returns the first position of equality or the position of the end-of-string terminator \0.

This built-in function is valid only when -qarch(-mcpu) is set to utilize the POWER9 technology and the altivec.h file is included.

Syntax

```
d=vec_first_match_index(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 113. Result and argument types

d	a	b
unsigned int	vector signed char	vector signed char
unsigned int	vector unsigned char	vector unsigned char
unsigned int	vector signed short	vector signed short
unsigned int	vector unsigned short	vector unsigned short
unsigned int	vector signed int	vector signed int
unsigned int	vector unsigned int	vector unsigned int

Result value

Returns the byte index of the position of the first equal element between a and b or the position of an end-of-string terminator, *\0*.

If both of the following conditions are met,
- There is no equality between the corresponding elements of a and b;
- There is no end-of-string terminator;

returns the number of the elements of either a or b.

vec_first_mismatch_index

Purpose

Compares each set of the corresponding elements of the given vectors and returns the first position of inequality.

This built-in function is valid only when -qarch(-mcpu) is set to utilize the POWER9 technology and the altivec.h file is included.

Syntax

```
d=vec_first_mismatch_index(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 114. Result and argument types

d	a	b
unsigned int	vector signed char	vector signed char

d	a	b
unsigned int	vector unsigned char	vector unsigned char
unsigned int	vector signed short	vector signed short
unsigned int	vector unsigned short	vector unsigned short
unsigned int	vector signed int	vector signed int
unsigned int	vector unsigned int	vector unsigned int

Result value

Returns the byte index of the position of the first unequal element between a and b. If all the corresponding elements are equal between a and b, returns the number of the elements of either a or b.

vec_first_mismatch_or_eos_index

Purpose

Compares each set of the corresponding elements of the given vectors and returns the first position of inequality or the position of the end-of-string terminator \0.

This built-in function is valid only when `-qarch(-mcpu)` is set to utilize the POWER9 technology and the `altivec.h` file is included.

Syntax

```
d=vec_first_mismatch_index(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 115. Result and argument types

d	a	b
unsigned int	vector signed char	vector signed char
unsigned int	vector unsigned char	vector unsigned char
unsigned int	vector signed short	vector signed short
unsigned int	vector unsigned short	vector unsigned short
unsigned int	vector signed int	vector signed int
unsigned int	vector unsigned int	vector unsigned int

Result value

Returns the byte index of the first position of the unequal element between a and b or the position of an end-of-string terminator, \0.

If both of the following conditions are met,
- There is no unequality between the corresponding elements of a and b;
- There is no end-of-string terminator;

returns the number of the elements of either a or b.

vec_gbb

Purpose

Performs a gather-bits-by-bytes operation on the input.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

d=vec_gbb(a)

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 116. Result and argument types

d	a
vector unsigned long long	vector unsigned long long
vector signed long long	vector signed long long

Result value

Each doubleword element of the result is set as follows: Let x(i) (0 <= i < 8) denote the byte elements of the corresponding input doubleword element, with x(7) the most significant byte. For each pair of i and j (0 <= i < 8, 0 <= j < 8), the jth bit of the ith byte element of the result is set to the value of the ith bit of the jth byte element of the input.

vec_insert

Purpose

Returns a copy of the vector b with the value of its element c replaced by a.

This built-in function is valid only when you include the `altivec.h` file.

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.

Table 117. Result and argument types

d	a	b	c
vector signed char	signed char	vector signed char	signed int
vector unsigned char	unsigned char	vector bool char	
		vector unsigned char	
vector signed short	signed short	vector signed short	
vector unsigned short	unsigned short	vector bool short	
		vector unsigned short	
vector signed int	signed int	vector signed int	
vector unsigned int	unsigned int	vector bool int	
		vector unsigned int	
vector signed long long	signed long long	vector signed long long	
vector unsigned long long	unsigned long long	vector bool long long	
		vector unsigned long long	
vector float	float	vector float	
vector double	double	vector double	

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.

vec_insert_exp

Purpose

Returns a vector that combines the exponents of elements from one vector with the signs and the significands of elements from another vector.

This built-in function is valid only when -qarch(-mcpu) is set to utilize the POWER9 technology and the altivec.h file is included.

Syntax

d=vec_insert_exp(a, b)

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 118. Types of the returned value and function arguments

d	a	b
vector float	vector unsigned int	vector unsigned int
vector float	vector float	vector unsigned int
vector double	vector unsigned long long	vector unsigned long long

Table 118. Types of the returned value and function arguments (continued)

d	a	b
vector double	vector double	vector unsigned long long

Result value

Each element of the result is generated by combining the exponent of the corresponding element of b with the sign and the significand of the corresponding element of a.

Note: The inserted exponent of b is treated as a biased exponent in accordance with the format specified by *IEEE Standard for Binary Floating-Point Arithmetic, IEEE Std. 754*, without further processing.

vec_ld

Purpose

Loads a vector from the given memory address.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_ld(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 119. Data type of function returned value and arguments

d	a	b
vector unsigned int	int	const unsigned long*
vector signed int		const signed long*

Table 119. Data type of function returned value and arguments (continued)

d	a	b
vector unsigned char	long	const vector unsigned char*
		const unsigned char*
vector signed char		const vector signed char*
		const signed char*
vector unsigned short		const vector unsigned short*
		const unsigned short*
vector signed short		const vector signed short*
		const signed short*
vector unsigned int		const vector unsigned int*
		const unsigned int*
vector signed int		const vector signed int*
		const signed int*
vector float		const vector float*
		const float*
vector bool int		const vector bool int*
vector bool char		const vector bool char*
vector bool short		const vector bool short*
vector pixel		const vector pixel*

Result value

The value of a is added to the address that is specified by b, and the sum is truncated to a multiple of 16 bytes. The result is the content of the 16 bytes of memory starting at this address.

vec_lde
Purpose

Loads an element from a given memory address into a vector.

This built-in function is valid only when you include the `altivec.h` file.

Syntax
```
d=vec_lde(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 120. Types of the returned value and function arguments

d	a	b
vector signed char	Any integral type	const signed char *
vector unsigned char		const unsigned char *
vector signed short		const short *
vector unsigned short		const unsigned short *
vector signed int		const int *
vector unsigned int		const unsigned int *
vector float		const float *

Result value

The effective address is the sum of a and the address specified by b, truncated to a multiple of the size in bytes of an element of the result vector. The contents of memory at the effective address are loaded into the result vector at the byte offset corresponding to the four least significant bits of the effective address. The remaining elements of the result vector are undefined.

vec_ldl
Purpose

Loads a vector from a given memory address, and marks the cache line containing the data as Least Recently Used.

This built-in function is valid only when you include the `altivec.h` file.

Syntax
`d=vec_ldl(a, b)`

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 121. Types of the returned value and function arguments

d	a	b
vector bool char	Any integral type	const vector bool char *
vector signed char		const signed char *
		const vector signed char *
vector unsigned char		const unsigned char *
		const vector unsigned char *
vector bool short		const vector bool short *
vector signed short		const signed short *
		const vector signed short *
vector unsigned short		const unsigned short *
		const vector unsigned short *
vector bool int		const vector bool int *
vector signed int		const signed int *
		const vector signed int *
vector unsigned int		const unsigned int *
		const vector unsigned int *
vector float		const float *
		const vector float *
vector pixel		const vector pixel *

Result value

a is added to the address specified by b, and the sum is truncated to a multiple of 16 bytes. The result is the contents of the 16 bytes of memory starting at this address. This data is marked as Least Recently Used.

vec_load_splats
Purpose

Loads a 4-byte element from the memory address specified by the displacement a and the pointer b and then splats it onto a vector.

This built-in function is valid only when -qarch(-mcpu) is set to utilize the POWER9 technology and the altivec.h file is included.

Syntax

d=vec_load_splats(a, b)

Result and argument types

The following table describes the types of the result and the function arguments.

Table 122. Result and argument types

d	a	b
vector signed int	signed long long	signed int *
vector signed int	unsigned long long	signed int *

Table 122. Result and argument types (continued)

d	a	b
vector unsigned int	signed long long	unsigned int *
vector unsigned int	unsigned long long	unsigned int *
vector float	signed long long	float *
vector float	unsigned long long	float *

Result value

The result is a vector with each element set to the 4-byte element from the effective address calculated by adding the displacement provided by a and the pointer b. The effective address is not truncated to a multiple of 16 bytes.

vec_loge
Purpose

Returns a vector containing estimates of the base-2 logarithms of the corresponding elements of the given vector.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_loge(a)
```

Result and argument types

The type of d and a must be `vector float`.

Result value

Each element of the result contains the estimated value of the base-2 logarithm of the corresponding element of a.

vec_lvsl

Purpose

Returns a vector useful for aligning non-aligned data.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_lvsl(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 123. Data type of function returned value and arguments

d	a	b
vector unsigned char	int	unsigned long*
		long*
	long	unsigned char*
		signed char*
		unsigned short*
		short*
		unsigned int*
		int*
		float*

Result value

The first element of the result vector is the sum of the value of a and the address that is specified by b, modulo 16. The value of each successive element is the value of the previous element plus 1.

vec_lvsr

Purpose

Returns a vector useful for aligning non-aligned data.

This built-in function is valid only when you include the altivec.h file.

Syntax

d=vec_lvsr(a, b)

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 124. Data type of function returned value and arguments

d	a	b
vector unsigned char	int	unsigned long*
		long*
	long	unsigned char*
		signed char*
		unsigned short*
		short*
		unsigned int*
		int*
		float*

Result value

The effective address is the sum of the value of a and the address that is specified by b, modulo 16. The first element of the result vector contains the value 16 minus the effective address. The value of each successive element is the value of the previous element plus 1.

vec_madd

Purpose

Returns a vector containing the results of performing a fused multiply-add operation on each corresponding set of elements of three given vectors.

This built-in function is valid only when you include the altivec.h file.

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.

Table 125. Types of the returned value and the function arguments

d	a	b	c
The same type as argument a	vector float	The same type as argument a	The same type as argument a
	vector double		

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.

vec_madds
Purpose

Returns a vector containing the results of performing a saturated multiply-high-and-add operation on each corresponding set of elements of three given vectors.

This built-in function is valid only when you include the altivec.h file.

Syntax

d=vec_madds(a, b, c)

Result and argument types

The type of d, a, b, and c must be vector signed short.

Result value

For each element of the result, the value is produced in the following way: the values of the corresponding elements of a and b are multiplied. The value of the 17 most significant bits of this product is then added, using 16-bit-saturated addition, to the value of the corresponding element of c.

vec_max

Purpose

Returns a vector containing the maximum value from each set of corresponding elements of the given vectors.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

`d=vec_max(a, b)`

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 126. Result and argument types

d	a	b
vector signed char	vector bool char	vector signed char
	vector signed char	vector signed char
		vector bool char
vector unsigned char	vector bool char	vector unsigned char
	vector unsigned char	vector unsigned char
		vector bool char
vector signed short	vector bool short	vector signed short
	vector signed short	vector signed short
		vector bool short
vector unsigned short	vector bool short	vector unsigned short
	vector unsigned short	vector unsigned short
		vector bool short
vector signed int	vector bool int	vector signed int
	vector signed int	vector signed int
		vector bool int
vector unsigned int	vector bool int	vector unsigned int
	vector unsigned int	vector unsigned int
		vector bool int
vector float	vector float	vector float
vector double	vector double	vector double
vector signed long long	vector signed long long	vector signed long long
vector unsigned long long	vector unsigned long long	vector unsigned long long

Table 126. Result and argument types (continued)

d	a	b
vector bool long long	vector bool long long	vector bool long long

Result value

The value of each element of the result is the maximum of the values of the corresponding elements of a and b.

vec_mergee

Purpose

Merges the values of even-numbered elements of two vectors.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_mergee(a,b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 127. Result and argument types

d	a	b
The same type as argument a	vector bool int	The same type as argument a
	vector signed int	
	vector unsigned int	

Result value

Assume that the elements of each vector are numbered beginning with zero. The even-numbered elements of the result are obtained, in order, from the even-numbered elements of a. The odd-numbered elements of the result are obtained, in order, from the even-numbered elements of b.

Related information
"vec_mergeo" on page 437

vec_mergeh
Purpose

Merges the most significant halves of two vectors.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_mergeh(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 128. Result and argument types

d	a	b
The same type as argument a	vector bool char	The same type as argument a
	vector signed char	
	vector unsigned char	
	vector bool short	
	vector signed short	
	vector unsigned short	
	vector bool int	
	vector signed int	
	vector unsigned int	
	vector bool long long	
	vector signed long long	
	vector unsigned long long	
	vector float	
	vector double	

Result value

Assume that the elements of each vector are numbered beginning with 0. The even-numbered elements of the result are taken, in order, from the high elements of a. The odd-numbered elements of the result are taken, in order, from the high elements of b.

Related reference:

"-maltivec (-qaltivec)" on page 129

"vec_mergel"

Related information:

Vector element order toggling

vec_mergel
Purpose

Merges the least significant halves of two vectors.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_mergel(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 129. Result and argument types

d	a	b
The same type as argument a	vector bool char	The same type as argument a
	vector signed char	
	vector unsigned char	
	vector bool short	
	vector signed short	
	vector unsigned short	
	vector bool int	
	vector signed int	
	vector unsigned int	
	vector bool long long	
	vector signed long long	
	vector unsigned long long	
	vector float	
	vector double	

Result value

Assume that the elements of each vector are numbered beginning with 0. The even-numbered elements of the result are taken, in order, from the low elements of a. The odd-numbered elements of the result are taken, in order, from the low elements of b.

Related reference:

"-maltivec (-qaltivec)" on page 129

"vec_mergeh" on page 435

Related information:

Vector element order toggling

vec_mergeo

Purpose

Merges the values of odd-numbered elements of two vectors.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_mergeo(a,b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 130. Result and argument types

d	a	b
The same type as argument a	vector bool int	The same type as argument a
	vector signed int	
	vector unsigned int	

Result value

Assume that the elements of each vector are numbered beginning with zero. The even-numbered elements of the result are obtained, in order, from the odd-numbered elements of a. The odd-numbered elements of the result are obtained, in order, from the odd-numbered elements of b.

Related information

"vec_mergee" on page 435

vec_mfvscr
Purpose

Copies the contents of the Vector Status and Control Register into the result vector.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_mfvscr()
```

Result and argument types

This function does not have any arguments. The result is of type `vector unsigned short`.

Result value

The high-order 16 bits of the VSCR are copied into the seventh element of the result. The low-order 16 bits of the VSCR are copied into the eighth element of the result. All other elements are set to zero.

vec_min
Purpose

Returns a vector containing the minimum value from each set of corresponding elements of the given vectors.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_min(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 131. Result and argument types

d	a	b
vector signed char	vector bool char	vector signed char
	vector signed char	vector signed char
		vector bool char
vector unsigned char	vector bool char	vector unsigned char
	vector unsigned char	vector unsigned char
		vector bool char
vector signed short	vector bool short	vector signed short
	vector signed short	vector signed short
		vector bool short
vector unsigned short	vector bool short	vector unsigned short
	vector unsigned short	vector unsigned short
		vector bool short
vector signed int	vector bool int	vector signed int
	vector signed int	vector signed int
		vector bool int
vector unsigned int	vector bool int	vector unsigned int
	vector unsigned int	vector unsigned int
		vector bool int
vector float	vector float	vector float
vector double	vector double	vector double
vector signed long long	vector signed long long	vector signed long long
vector unsigned long long	vector unsigned long long	vector unsigned long long
vector bool long long	vector bool long long	vector bool long long

Result value

The value of each element of the result is the minimum of the values of the corresponding elements of a and b.

vec_mladd
Purpose

Returns a vector containing the results of performing a saturated multiply-low-and-add operation on each corresponding set of elements of three given vectors.

This built-in function is valid only when you include the altivec.h file.

Syntax

```
d=vec_mladd(a, b, c)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 132. Types of the returned value and function arguments

d	a	b	c
vector signed short	vector signed short	vector signed short	vector signed short
	vector signed short	vector unsigned short	vector unsigned short
	vector unsigned short	vector signed short	vector signed short
vector unsigned short	vector unsigned short	vector unsigned short	vector unsigned short

Result value

The value of each element of the result is the value of the least significant 16 bits of the product of the values of the corresponding elements of a and b, added to the value of the corresponding element of c.

The addition is performed using modular arithmetic.

vec_mradds

Purpose

Returns a vector containing the results of performing a saturated multiply-high-round-and-add operation for each corresponding set of elements of the given vectors.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_mradds(a, b, c)
```

Result and argument types

The type of d, a, b, and c must be `vector unsigned short`.

Result value

For each element of the result, the value is produced in the following way: the values of the corresponding elements of a and b are multiplied and rounded such that the 15 least significant bits are 0. The value of the 17 most significant bits of this rounded product is then added, using 16-bit-saturated addition, to the value of the corresponding element of c.

vec_msub

Purpose

Returns a vector containing the results of performing a multiply-subtract operation using the given vectors.

This built-in function is valid only when you include the `altivec.h` file.

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.

Table 133. Result and argument types

d	a	b	c
vector float	vector float	vector float	vector float
vector double	vector double	vector double	vector double

Result value

This function multiplies each element in a by the corresponding element in b and then subtracts the corresponding element in c from the result.

vec_msum
Purpose

Returns a vector containing the results of performing a multiply-sum operation using given vectors.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_msum(a, b, c)
```

Result and argument types

The following tables describe the types of the returned value and the function arguments.

Table 134. Types of the returned value and function arguments

d	a	b	c
vector signed int	vector signed char	vector unsigned char	vector signed int
vector unsigned int	vector unsigned char	vector unsigned char	vector unsigned int
vector signed int	vector signed short	vector signed short	vector signed int
vector unsigned int	vector unsigned short	vector unsigned short	vector unsigned int

Result value

For each element n of the result vector, the value is obtained as follows:

* If a is of type `vector signed char` or `vector unsigned char`, multiply element p of a by element p of b where p is from 4n to 4n+3, and then add the sum of these products and element n of c.

```
d[0] = a[0]*b[0]    + a[1]*b[1]    + a[2]*b[2]    + a[3]*b[3]    + c[0]
d[1] = a[4]*b[4]    + a[5]*b[5]    + a[6]*b[6]    + a[7]*b[7]    + c[1]
d[2] = a[8]*b[8]    + a[9]*b[9]    + a[10]*b[10]  + a[11]*b[11]  + c[2]
d[3] = a[12]*b[12]  + a[13]*b[13]  + a[14]*b[14]  + a[15]*b[15]  + c[3]
```

- If a is of type `vector signed short` or `vector unsigned short`, multiply element p of a by element p of b where p is from 2n to 2n+1, and then add the sum of these products and element n of c.

```
d[0] = a[0]*b[0] + a[1]*b[1] + c[0]
d[1] = a[2]*b[2] + a[3]*b[3] + c[1]
d[2] = a[4]*b[4] + a[5]*b[5] + c[2]
d[3] = a[6]*b[6] + a[7]*b[7] + c[3]
```

All additions are performed by using 32-bit modular arithmetic.

vec_msums
Purpose

Returns a vector containing the results of performing a saturated multiply-sum operation using the given vectors.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

`d=vec_msums(a, b, c)`

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 135. Types of the returned value and function arguments

d	a	b	c
vector signed int	vector signed short	vector signed short	vector signed int
vector unsigned int	vector unsigned short	vector unsigned short	vector unsigned int

Result value

For each element n of the result vector, the value is obtained in the following way: multiply element p of a by element p of b, where p is from 2n to 2n+1; and then add the sum of these products to element n of c. All additions are performed by using 32-bit saturated arithmetic.

vec_mtvscr
Purpose

Copies the given value into the Vector Status and Control Register.

The low-order 32 bits of a are copied into the VSCR.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

`vec_mtvscr(a)`

Result and argument types

This function does not return any value. `a` is of any of the following types:

- vector bool char
- vector signed char
- vector unsigned char
- vector bool short
- vector signed short
- vector unsigned short
- vector bool int
- vector signed int
- vector unsigned int
- vector pixel

vec_mul

Purpose

Returns a vector containing the results of performing a multiply operation using the given vectors.

Note: For integer and unsigned vectors, this function emulates the operation.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_mul(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 136. Result and argument types

d	a	b
The same type as argument a	vector signed char	The same type as argument a
	vector unsigned char	
	vector signed short	
	vector unsigned short	
	vector signed int	
	vector unsigned int	
	vector signed long long	
	vector unsigned long long	
	vector float	
	vector double	

Result value

This function multiplies corresponding elements in the given vectors and then assigns the result to corresponding elements in the result vector.

vec_mule
Purpose

Returns a vector containing the results of multiplying every second set of corresponding elements of the given vectors, beginning with the first element.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

`d=vec_mule(a, b)`

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 137. Types of the returned value and function arguments

d	a	b
vector signed short	vector signed char	vector signed char
vector unsigned short	vector unsigned char	vector unsigned char
vector signed int	vector signed short	vector signed short
vector unsigned int	vector unsigned short	vector unsigned short
vector signed long long	vector signed int	vector signed int
vector unsigned long long	vector unsigned int	vector unsigned int

Result value

Assume that the elements of each vector are numbered beginning with 0. For each element n of the result vector, the value is the product of the value of element 2n of a and the value of element 2n of b.

vec_mulo
Purpose

Returns a vector containing the results of multiplying every second set of corresponding elements of the given vectors, beginning with the second element.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

`d=vec_mulo(a, b)`

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 138. Types of the returned value and function arguments

d	a	b
vector signed short	vector signed char	vector signed char
vector unsigned short	vector unsigned char	vector unsigned char
vector signed int	vector signed short	vector signed short
vector unsigned int	vector unsigned short	vector unsigned short
vector signed long long	vector signed int	vector signed int
vector unsigned long long	vector unsigned int	vector unsigned int

Result value

Assume that the elements of each vector are numbered beginning with 0. For each element n of the result vector, the value is the product of the value of element 2n+1 of a and the value of element 2n+1 of b.

vec_nabs

Purpose

Returns a vector containing the results of performing a negative-absolute operation using the given vector.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_nabs(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 139. Result and argument types

d	a
vector float	vector float
vector double	vector double

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.

vec_nand

Purpose

Performs a bitwise negated-and operation on the input vectors.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_nand(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 140. Types of the returned value and function arguments

d	a	b
vector signed char	vector signed char	vector signed char
		vector bool char
vector unsigned char	vector unsigned char	vector unsigned char
		vector bool char
vector signed char	vector bool char	vector signed char
vector unsigned char		vector unsigned char
vector bool char		vector bool char
vector signed short	vector signed short	vector signed short
		vector bool short
vector unsigned short	vector unsigned short	vector unsigned short
		vector bool short
vector signed short	vector bool short	vector signed short
vector unsigned short		vector unsigned short
vector bool short		vector bool short
vector signed int	vector signed int	vector signed int
		vector bool int
vector unsigned int	vector unsigned int	vector unsigned int
		vector bool int
vector signed int	vector bool int	vector signed int
vector unsigned int		vector unsigned int
vector bool int		vector bool int
vector float		vector float
vector signed long long	vector signed long long	vector signed long long
		vector bool long long
vector unsigned long long	vector unsigned long long	vector unsigned long long
		vector bool long long
vector signed long long	vector bool long long	vector signed long long
vector unsigned long long		vector unsigned long long
vector bool long long		vector bool long long
vector double		vector double
vector float	vector float	vector bool int
		vector float
vector double	vector double	vector long long
		vector double

Result value

Each bit of the result is set to the result of the bitwise operation !(a & b) of the corresponding bits of a and b. For 0 <= i < 128, bit i of the result is set to 0 only if the ith bits of both a and b are 1.

vec_ncipher_be

Purpose

Performs one round of the AES inverse cipher operation, as defined in *Federal Information Processing Standards Publication 197 (FIPS-197)*, on an intermediate state a by using a given round key b.

This built-in function is valid only when you include the altivec.h file.

Syntax

d=vec_ncipher_be(a, b)

Result and argument types

The type of d, a, and b must be vector unsigned char.

Result value

Returns the resulting intermediate state.

vec_ncipherlast_be

Purpose

Performs the final round of the AES inverse cipher operation, as defined in *Federal Information Processing Standards Publication 197 (FIPS-197)*, on an intermediate state a by using a given round key b.

This built-in function is valid only when you include the altivec.h file.

Syntax

d=vec_ncipherlast_be(a, b)

Result and argument types

The type of d, a, and b must be vector unsigned char.

Result value

Returns the resulting final state.

vec_nearbyint
Purpose

Returns a vector that contains the rounded values of the corresponding elements of the given vector.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_nearbyint(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a
vector float	vector float
vector double	vector double

Result value

Each element of the result contains the value of the corresponding element of `a`, rounded to the nearest representable floating-point integer, using IEEE round-to-nearest rounding. When an input element value is between two integer values, the result value with the larger absolute value is selected.

Related reference:

"vec_round" on page 466

vec_neg

Purpose

Returns a vector containing the negated value of the corresponding elements in the given vector.

Note: For `vector signed long long`, this function emulates the operation.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_neg(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 141. Result and argument types

d	a
The same type as argument a	vector signed char
	vector signed short
	vector signed int
	vector signed long long
	vector float
	vector double

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.

vec_nmadd

Purpose

Returns a vector containing the results of performing a negative multiply-add operation on the given vectors.

This built-in function is valid only when you include the `altivec.h` file.

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.

Table 142. Result and argument types

d	a	b	c
vector double	vector double	vector double	vector double
vector float	vector float	vector float	vector float

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.

vec_nmsub

Purpose

Returns a vector containing the results of performing a negative multiply-subtract operation on the given vectors.

This built-in function is valid only when you include the `altivec.h` file.

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.

Table 143. Result and argument types

d	a	b	c
vector float	vector float	vector float	vector float

Table 143. Result and argument types (continued)

d	a	b	c
vector double	vector double	vector double	vector double

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.

vec_nor

Purpose

Performs a bitwise NOR of the given vectors.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_nor(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 144. Result and argument types

d	a	b
vector bool char	vector bool char	vector bool char
vector signed char	vector bool char	vector signed char
	vector signed char	vector signed char
		vector bool char
vector unsigned char	vector bool char	vector unsigned char
	vector unsigned char	vector unsigned char
		vector bool char
vector bool short	vector bool short	vector vector bool short
vector signed short	vector bool short	vector signed short
	vector signed short	vector signed short
		vector bool short
vector unsigned short	vector bool short	vector unsigned short
	vector unsigned short	vector unsigned short
		vector bool short
vector bool int	vector bool int	vector bool int
vector signed int	vector bool int	vector signed int
	vector signed int	vector signed int
		vector bool int

Table 144. Result and argument types (continued)

d	a	b
vector unsigned int	vector bool int	vector unsigned int
	vector unsigned int	vector unsigned int
		vector bool int
vector bool long long	vector bool long long	vector bool long long
vector signed long long	vector signed long long	vector signed long long
vector unsigned long long	vector unsigned long long	vector unsigned long long
vector float	vector bool int	vector float
	vector float	vector bool int
vector double	vector double	vector double

Result value

The result is the bitwise NOR of `a` and `b`.

vec_or

Purpose

Performs a bitwise OR of the given vectors.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_or(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 145. Result and argument types

d	a	b
vector bool char	vector bool char	vector bool char
vector signed char	vector bool char	vector signed char
	vector signed char	vector signed char
		vector bool char
vector unsigned char	vector bool char	vector unsigned char
	vector unsigned char	vector unsigned char
		vector bool char
vector bool short	vector bool short	vector vector bool short
vector signed short	vector bool short	vector signed short
	vector signed short	vector signed short
		vector bool short

Table 145. Result and argument types (continued)

d	a	b
vector unsigned short	vector bool short	vector unsigned short
	vector unsigned short	vector unsigned short
		vector bool short
vector bool int	vector bool int	vector bool int
vector signed int	vector bool int	vector signed int
	vector signed int	vector signed int
		vector bool int
vector unsigned int	vector bool int	vector unsigned int
	vector unsigned int	vector unsigned int
		vector bool int
vector bool long long	vector bool long long	vector bool long long
vector signed long long	vector bool long long	vector signed long long
	vector signed long long	vector signed long long
		vector bool long long
vector unsigned long long	vector bool long long	vector unsigned long long
	vector unsigned long long	vector unsigned long long
		vector bool long long
vector float	vector bool int	vector float
	vector float	vector bool int
		vector float
vector double	vector bool long long	vector double
	vector double	vector bool long long
		vector double

Result value

The result is the bitwise OR of a and b.

vec_orc
Purpose

Performs a bitwise OR-with-complement operation of the input vectors.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_orc(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 146. Types of the returned value and function arguments

d	a	b
vector signed char	vector signed char	vector signed char
		vector bool char
vector unsigned char	vector unsigned char	vector unsigned char
		vector bool char
vector signed char	vector bool char	vector signed char
vector unsigned char		vector unsigned char
vector bool char		vector bool char
vector signed short	vector signed short	vector signed short
		vector bool short
vector unsigned short	vector unsigned short	vector unsigned short
		vector bool short
vector signed short	vector bool short	vector signed short
vector unsigned short		vector unsigned short
vector bool short		vector bool short
vector signed int	vector signed int	vector signed int
		vector bool int
vector unsigned int	vector unsigned int	vector unsigned int
		vector bool int
vector signed int	vector bool int	vector signed int
vector unsigned int		vector unsigned int
vector bool int		vector bool int
vector float		vector float
vector signed long long	vector signed long long	vector signed long long
		vector bool long long
vector unsigned long long	vector unsigned long long	vector unsigned long long
		vector bool long long
vector signed long long	vector bool long long	vector signed long long
vector unsigned long long		vector unsigned long long
vector bool long long		vector bool long long
vector double		vector double
vector float	vector float	vector bool int
		vector float
vector double	vector double	vector bool long long
		vector double

Result value

Each bit of the result is set to the result of the bitwise operation (a | ~b) of the corresponding bits of a and b. For 0 <= i < 128, bit i of the result is set to 1 only if the ith bit of a is 1 or the ith bit of b is 0.

vec_pack

Purpose

Packs information from each element of two vectors into the result vector.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

`d=vec_pack(a, b)`

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 147. Result and argument types

d	a	b
vector signed char	vector signed short	vector signed short
vector unsigned char	vector unsigned short	vector unsigned short
vector signed short	vector signed int	vector signed int
vector unsigned short	vector unsigned int	vector unsigned int
vector signed int	vector signed long long	vector signed long long
vector unsigned int	vector unsigned long long	vector unsigned long long
vector bool long long	vector bool long long	vector bool long long

Result value

The value of each element of the result vector is taken from the low-order half of the corresponding element of the result of concatenating a and b.

Related reference:

"-maltivec (-qaltivec)" on page 129

Related information:

Vector element order toggling

vec_packpx

Purpose

Packs information from each element of two vectors into the result vector.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

`d=vec_packpx(a, b)`

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 148. Types of the returned value and function arguments

d	a	b
vector pixel	vector unsigned int	vector unsigned int

Result value

The value of each element of the result vector is taken from the corresponding element of the result of concatenating a and b in the following way: the least significant bit of the high order byte is stored into the first bit of the result element; the most significant 5 bits of each of the remaining bytes are stored into the remaining portion of the result element.

```
d[i]   = a_i[7]  ||  a_i[8:12]  ||  a_i[16:20]  ||  a_i[24:28]
d[i+4] = b_i[7]  ||  b_i[8:12]  ||  b_i[16:20]  ||  b_i[24:28]
```

where *i* is 0, 1, 2, and 3.

vec_packs

Purpose

Packs information from each element of two vectors into the result vector, using saturated values.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_packs(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 149. Result and argument types

d	a	b
vector signed char	vector signed short	vector signed short
vector unsigned char	vector unsigned short	vector unsigned short
vector signed short	vector signed int	vector signed int
vector unsigned short	vector unsigned int	vector unsigned int
vector signed int	vector signed long long	vector signed long long
vector unsigned int	vector unsigned long long	vector unsigned long long

Result value

The value of each element of the result vector is the saturated value of the corresponding element of the result of concatenating a and b.

vec_packsu

Purpose

Packs information from each element of two vectors into the result vector by using saturated values.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

`d=vec_packsu(a, b)`

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 150. Result and argument types

d	a	b
vector unsigned char	vector signed short	vector signed short
	vector unsigned short	vector unsigned short
vector unsigned short	vector signed int	vector signed int
	vector unsigned int	vector unsigned int
vector unsigned int	vector signed long long	vector signed long long
	vector unsigned long long	vector unsigned long long

Result value

The value of each element of the result vector is the saturated value of the corresponding element of the result of concatenating a and b.

vec_parity_lsbb

Purpose

Returns a vector that computes parity on the least significant bit of each byte of each element of the given vector.

This built-in function is valid only when `-qarch(-mcpu)` is set to utilize the POWER9 technology and the `altivec.h` file is included.

Syntax

`d=vec_parity_lsbb(a)`

Result and argument types

The following table describes the types of the returned value and the function argument.

Table 151. Types of the returned value and function argument

d	a
vector unsigned int	vector signed int

Table 151. Types of the returned value and function argument (continued)

d	a
vector unsigned int	vector unsigned int
vector unsigned long long	vector signed long long
vector unsigned long long	vector unsigned long long

Result value

The value of each element of the result is the parity of the least significant bit of each byte of the corresponding element of a.

vec_perm

Purpose

Returns a vector that contains some elements of two vectors, in the order specified by a third vector.

This built-in function is valid only when you include the `altivec.h` file.

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.

Table 152. Result and argument types

d	a	b	c
The same type as argument a	vector signed int	The same type as argument a	vector unsigned char
	vector unsigned int		
	vector bool int		
	vector signed short		
	vector unsigned short		
	vector bool short		
	vector pixel		
	vector signed char		
	vector unsigned char		
	vector bool char		
	vector float		
	vector double		
	vector signed long long		
	vector unsigned long long		

Result value

Each byte of the result is selected by using the least significant five bits of the corresponding byte of c as an index into the concatenated bytes of a and b.

vec_permi
Purpose

Returns a vector by permuting and combining two eight-byte-long vector elements in a and b based on the value of c.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

`d=vec_permi(a, b, c)`

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 153. Result and argument types

d	a	b	c
vector bool long long	vector bool long long	vector bool long long	0–3
vector signed long long	vector signed long long	vector signed long long	
vector unsigned long long	vector unsigned long long	vector unsigned long long	
vector double	vector double	vector double	

Result value

If we use a[0] and a[1] to represent the first and second eight-byte-long elements in a, and use b[0] and b[1] for elements in b, then this function determines the elements in the result vector based on the binary value of c. This is illustrated as follows:

- 00 - a[0],b[0]
- 01 - a[0], b[1]
- 10 - a[1], b[0]
- 11 - a[1], b[1]

vec_pmsum_be

Purpose

Performs an exclusive-OR operation by implementing a polynomial addition on each even-odd pair of the polynomial multiplication result of the corresponding elements.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

`d=vec_pmsum_be(a, b)`

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 154. Types of the returned value and function arguments

d	a	b
vector unsigned short	vector unsigned char	vector unsigned char
vector unsigned int	vector unsigned short	vector unsigned short
vector unsigned long long	vector unsigned int	vector unsigned int

Result value

Each element i of the result vector is computed by an exclusive-OR operation of the polynomial multiplication of input elements $2*i$ of a and b and input elements $2*i + 1$ of a and b.

`d[i] =(a[2*i]*b[2*i]) ^ (a[2*i + 1]*b[2*i + 1])`

vec_popcnt
Purpose

Computes the population count (number of set bits) in each element of the input.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

`d=vec_popcnt(a)`

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 155. Result and argument types

d	a
vector unsigned char	vector signed char
	vector unsigned char
vector unsigned short	vector signed short
	vector unsigned short
vector unsigned int	vector signed int
	vector unsigned int
vector unsigned long long	vector signed long long
	vector unsigned long long

Result value

Each element of the result is set to the number of set bits in the corresponding element of the input.

vec_promote

Purpose

Returns a vector with a in element position b.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

`d=vec_promote(a, b)`

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 156. Result and argument types

d	a	b
vector signed char	signed char	signed int
vector unsigned char	unsigned char	
vector signed short	signed short	
vector unsigned short	unsigned short	
vector signed int	signed int	
vector unsigned int	unsigned int	
vector signed long long	signed long long	
vector unsigned long long	unsigned long	
vector float	float	
vector double	double	

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.

vec_re

Purpose

Returns a vector containing estimates of the reciprocals of the corresponding elements of the given vector.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_re(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 157. Result and argument types

d	a
vector float	vector float
vector double	vector double

Result value

Each element of the result contains the estimated value of the reciprocal of the corresponding element of a.

vec_recipdiv

Purpose

Returns a vector that contains the division of each elements of a by the corresponding elements of b, by performing reciprocal estimates and iterative refinement on the elements of b.

This built-in function is valid only when you include the altivec.h file.

Syntax

```
d=vec_recipdiv(a,b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a	b
vector float	vector float	vector float
vector double	vector double	vector double

Result value

Each element of the result contains the approximate division of each element of a by the corresponding element of b. Vector reciprocal estimates and iterative refinement on each element of b are used to improve the accuracy of the approximation.

> **Related information**
>
> "vec_re" on page 460
>
> "vec_div" on page 412

vec_revb

Purpose

Returns a vector that contains the bytes of the corresponding element of the argument in the reverse byte order.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

`d=vec_revb(a)`

Result and argument types

The following table describes the types of the returned value and the function argument.

Table 158. Result and argument types

d	a
The same type as argument `a`	vector signed char
	vector unsigned char
	vector signed short
	vector unsigned short
	vector signed int
	vector unsigned int
	vector signed long long
	vector unsigned long long
	vector float
	vector double

Result value

Each element of the result contains the bytes of the corresponding element of `a` in the reverse byte order.

vec_reve

Purpose

Returns a vector that contains the elements of the argument in the reverse element order.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

`d=vec_reve(a)`

Result and argument types

The following table describes the types of the returned value and the function argument.

Table 159. Result and argument types

d	a
The same type as argument a	vector signed char
	vector unsigned char
	vector signed short
	vector unsigned short
	vector signed int
	vector unsigned int
	vector signed long long
	vector unsigned long long
	vector float
	vector double

Result value

The result contains the elements of a in the reverse element order.

vec_rint
Purpose

Returns a vector by rounding every single-precision or double-precision floating-point element of the given vector to a floating-point integer.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_rint(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a
vector float	vector float
vector double	vector double

Related reference:
"vec_roundc" on page 466

vec_rl
Purpose

Rotates each element of a vector left by a given number of bits.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_rl(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 160. Result and argument types

d	a	b
The same type as argument a	vector signed char	The same type as argument a
	vector unsigned char	
	vector signed short	
	vector unsigned short	
	vector signed int	
	vector unsigned int	
	vector signed long long	
	vector unsigned long long	

Result value

Each element of the result is obtained by rotating the corresponding element of a left by the number of bits specified by the corresponding element of b.

vec_rlmi

Purpose

Returns a vector that contains each element of the given vector rotated left and inserted under a mask into another vector.

This built-in function is valid only when -qarch(-mcpu) is set to utilize the POWER9 technology and the altivec.h file is included.

Syntax

```
d=vec_rlmi(a, b, c)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 161. Types of the returned value and function arguments

d	a	b	c
vector unsigned int	vector unsigned int	vector unsigned int	vector unsigned int
vector unsigned long long	vector unsigned long long	vector unsigned long long	vector unsigned long long

Result value

When vec_rlmi is called, each element of a is rotated left. The rotation count is specified by bits 27-31 of c. After the rotation, a is inserted under a mask into b.

Bits 11-15 of `c` contain the mask beginning (mb), and bits 19-23 of `c` contain the mask end (me). The mask is generated by setting all bits from bit mb through bit me to 1 and all other bits set to 0.

Note: You cannot specify an all-zero mask.

vec_rlnm

Purpose

Returns a vector that contains each element of the given vector rotated left and intersected with a mask.

This built-in function is valid only when `-qarch(-mcpu)` is set to utilize the POWER9 technology and the `altivec.h` file is included.

Syntax

```
d=vec_rlnm(a, b, c)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 162. Types of the returned value and function arguments

d	a	b	c
vector unsigned int	vector unsigned int	vector unsigned int	vector unsigned int
vector unsigned long long	vector unsigned long long	vector unsigned long long	vector unsigned long long

Result value

When vec_rlnm is called, each element of `a` is rotated left and intersected with a mask.

Each element of `b` contains the rotation count for the corresponding element of `a`. The low-order byte of each element is used, and you need to set other bytes to zero.

Each element of `c` contains the mask beginning and the mask end for the corresponding element of `a`. The mask end is in the low-order byte, the mask beginning is in the next higher byte, and you need to set other bytes to zero.

Note: You cannot specify an all-zero mask.

When the data type of the arguments is `vector unsigned int`, no more than 5 bits of each byte should be set for the rotation or the mask beginning and the end; when the data type is `vector unsigned long long`, no more than 6 bits of each byte should be set.

vec_round

Purpose

Returns a vector containing the rounded values of the corresponding elements of the given vector.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

`d=vec_round(a)`

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 163. Result and argument types

d	a
vector float	vector float
vector double	vector double

Result value

Each element of the result contains the value of the corresponding element of `a`, rounded to the nearest representable floating-point integer, using IEEE round-to-nearest rounding.

vec_roundc

Purpose

Returns a vector by rounding every single-precision or double-precision floating-point element in the given vector to integer.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

`d=vec_roundc(a)`

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 164. Result and argument types

d	a
vector float	vector float
vector double	vector double

Related information

"vec_rint" on page 463

vec_roundm

Purpose

Returns a vector containing the largest representable floating-point integer values less than or equal to the values of the corresponding elements of the given vector.

Note: `vec_roundm` is another name for `vec_floor`.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_roundm(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 165. Result and argument types

d	a
vector float	vector float
vector double	vector double

Related reference:
"vec_floor" on page 422

vec_roundp

Purpose

Returns a vector containing the smallest representable floating-point integer values greater than or equal to the values of the corresponding elements of the given vector.

Note: `vec_roundp` is another name for `vec_ceil`.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_roundp(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 166. Result and argument types

d	a
vector float	vector float
vector double	vector double

Related reference:
"vec_ceil" on page 398

vec_roundz

Purpose

Returns a vector containing the truncated values of the corresponding elements of the given vector.

Note: vec_roundz is another name for vec_trunc.

This built-in function is valid only when you include the altivec.h file.

Syntax

d=vec_roundz(a)

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 167. Result and argument types

d	a
vector float	vector float
vector double	vector double

Result value

Each element of the result contains the value of the corresponding element of a, truncated to an integral value.

Related reference:

"vec_trunc" on page 495

vec_rsqrt

Purpose

Returns a vector that contains a refined approximation of the reciprocal square roots of the corresponding elements of the given vector.

This built-in function is valid only when you include the altivec.h file.

Syntax

d=vec_rsqrt(a)

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a
vector float	vector float
vector double	vector double

Result value

Each element of the result contains a refined approximation of the reciprocal square root of the corresponding element of a.

Related reference:

"vec_rsqrte"

vec_rsqrte

Purpose

Returns a vector that contains the estimate value of the reciprocal square roots of the corresponding elements of the given vector.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

`d=vec_rsqrte(a)`

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 168. Result and argument types

d	a
vector float	vector float
vector double	vector double

Result value

Each element of the result contains the estimated value of the reciprocal square root of the corresponding element of a.

vec_sbox_be

Purpose

Performs the SubBytes operation, as defined in *Federal Information Processing Standards FIPS-197*, on a given state a.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

`d=vec_sbox_be(a)`

Result and argument types

The type of d and a must be `vector unsigned char`.

Result value

Returns the result of the SubBytes operation.

vec_sel

Purpose

Returns a vector containing the value of either a or b depending on the value of c.

This built-in function is valid only when you include the `altivec.h` file.

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.

Table 169. Result and argument types

d	a	b	c
The same type as argument b	The same type as argument b	vector bool char	vector bool char
			vector unsigned char
		vector signed char	vector bool char
			vector unsigned char
		vector unsigned char	vector bool char
			vector unsigned char
		vector bool short	vector bool short
			vector unsigned short
		vector signed short	vector bool shot
			vector unsigned short
		vector unsigned short	vector bool short
			vector unsigned short
		vector bool int	vector bool int
			vector unsigned int
		vector signed int	vector bool int
			vector unsigned int
		vector unsigned int	vector bool int
			vector unsigned int
		vector bool long long	vector bool long long
			vector unsigned long long
		vector signed long long	vector bool long long
			vector unsigned long long
		vector unsigned long long	vector bool long long
			vector unsigned long long
		vector float	vector bool int
			vector unsigned int
		vector double	vector bool long long
			vector unsigned long long

Result value

Each bit of the result vector has the value of the corresponding bit of a if the corresponding bit of c is 0, or the value of the corresponding bit of b otherwise.

vec_shasigma_be

Purpose

Performs a secure hash computation in accordance with *Federal Information Processing Standards FIPS-180-3*, which is a specification for the Secure Hash Standard.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_shasigma_be(a, b, c)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 170. Types of the returned value and function arguments

d	a	b[1]	c[2]
vector unsigned int	vector unsigned int	const int	const int
vector unsigned long long	vector unsigned long long	const int	const int

Notes:

1. b selects the function type, which can be either lowercase sigma (σ) or uppercase sigma ($\sum$). The argument must be a constant expression with a value of 0 or 1.

2. c selects the function subtype, which can be either sigma-0 ($\sigma 0$ or $\sum 0$) or sigma-1 ($\sigma 1$ or $\sum 1$). The argument must be a constant expression with a value in the range 0 - 15 inclusive.

Result value

- If a is of type `vector unsigned int`, for each element i ($i = 0,1,2,3$) of a, element i of the returned value is the result of the following SHA-256 function:
  - $\sigma 0(x[i])$, if b is 0 and bit i of the 4-bit c is 0
  - $\sigma 1(x[i])$, if b is 0 and bit i of the 4-bit c is 1
  - $\sum 0(x[i])$, if b is nonzero and bit i of the 4-bit c is 0
  - $\sum 1(x[i])$, if b is nonzero and bit i of the 4-bit c is 1
- If a is of type `vector unsigned long long`, for each element i ($i = 0,1$) of a, element i of the returned value is the result of the following SHA-512 function:
  - $\sigma 0(x[i])$, if b is 0 and bit $2*i$ of the 4-bit c is 0
  - $\sigma 1(x[i])$, if b is 0 and bit $2*i$ of the 4-bit c is 1
  - $\sum 0(x[i])$, if b is nonzero and bit $2*i$ of the 4-bit c is 0
  - $\sum 1(x[i])$, if b is nonzero and bit $2*i$ of the 4-bit c is 1

vec_sl

Purpose

Performs a left shift for each element of a vector.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_sl(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 171. Result and argument types

d	a	b
vector signed char	vector signed char	vector unsigned char
vector unsigned char	vector unsigned char	
vector signed short	vector signed short	vector unsigned short
vector unsigned short	vector unsigned short	
vector signed int	vector signed int	vector unsigned int
vector unsigned int	vector unsigned int	
vector signed long long	vector signed long long	vector unsigned long long
vector unsigned long long	vector unsigned long long	

Result value

Each element of the result vector is the result of left shifting the corresponding element of a by the number of bits specified by the value of the corresponding element of b, modulo the number of bits in the element. The bits that are shifted out are replaced by zeroes.

vec_sld

Purpose

Left shifts two concatenated vectors by a given number of bytes.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_sld(a, b, c)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 172. Types of the returned value and function arguments

d	a	b	c[1]
The same type as argument a	vector signed char	The same type as argument a	unsigned int
	vector unsigned char		
	vector signed short		
	vector unsigned short		
	vector signed int		
	vector unsigned int		
	vector float		
	vector pixel		

Note:

1. c must be an unsigned literal with a value in the range 0 - 15 inclusive.

Result value

The result is the most significant 16 bytes obtained by concatenating a and b, and shifting left by the number of bytes specified by c.

vec_sldw

Purpose

Left shifts two concatenated vectors by a given number of 4 bytes.

This built-in function is valid only when you include the `altivec.h` file.

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.

Table 173. Result and argument types

d	a	b	c
The same type as argument a	vector bool char	The same type as argument a	0–3
	vector signed char		
	vector unsigned char		
	vector bool short		
	vector signed short		
	vector unsigned short		
	vector bool int		
	vector signed int		
	vector unsigned int		
	vector bool long long		
	vector signed long long		
	vector unsigned long long		
	vector float		
	vector double		

Result value

The result is the most significant 16 bytes obtained by concatenating a and b, and shifting left by the number of 4 bytes specified by c.

vec_sll

Purpose

Left shifts a vector by a given number of bits.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_sll(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 174. Types of the returned value and function arguments

d	a	b[1]
The same type as argument a	vector bool char	Any of the following types: vector unsigned char vector unsigned short vector unsigned int
	vector signed char	
	vector unsigned char	
	vector bool short	
	vector signed short	
	vector unsigned short	
	vector bool int	
	vector signed int	
	vector unsigned int	
	vector pixel	

Note:

1. The least significant three bits of all byte elements in b must be the same.

Result value

The result is produced by shifting the contents of a left by the number of bits specified by the last three bits of the last element of b. The bits that are shifted out are replaced by zeroes.

vec_slo
Purpose

Left shifts a vector by a given number of bytes.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_slo(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 175. Types of the returned value and function arguments

d	a	b
The same type as argument a	vector signed char	Any of the following types: vector signed char vector unsigned char
	vector unsigned char	
	vector signed short	
	vector unsigned short	
	vector signed int	
	vector unsigned int	
	vector float	
	vector pixel	

Result value

The result is produced by shifting the contents of a left by the number of bytes specified by bits 121 through 124 of b. The bits that are shifted out are replaced by zeroes.

vec_slv

Purpose

Left shifts the elements of a given vector by a given number of bits.

This built-in function is valid only when -qarch(-mcpu) is set to utilize the POWER9 technology and the altivec.h file is included.

Syntax

d=vec_slv(a, b)

Result and argument types

The result and arguments are all of the vector unsigned char type.

Result value

Suppose the following information. Element i of the result vector contains the same bit string as bit S_i - bit S_{i+7} of X_i.

- S_i (0<=i<=15) denotes the value in the three least-significant bits of element i of b.
- X_i (0<=i<=14) denotes the halfword formed by concatenating elements i and $i+1$ of a.
- X_{15} denotes the halfword formed by concatenating element 15 of a and a zero byte.

vec_splat

Purpose

Returns a vector of which all the elements are set to a given value.

This built-in function is valid only when you include the altivec.h file.

Syntax

d=vec_splat(a, b)

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 176. Result and argument types

d	a	b
The same type as argument a	vector bool char	0 - 15
	vector signed char	0 - 15
	vector unsigned char	0 - 15
	vector bool short	0 - 7
	vector signed short	0 - 7
	vector unsigned short	0 - 7
	vector bool int	0 - 3
	vector signed int	0 - 3
	vector unsigned int	0 - 3
	vector bool long long	0 - 1
	vector signed long long	0 - 1
	vector unsigned long long	0 - 1
	vector float	0 - 3
	vector double	0 - 1

Result value

Each element of the result contains the value of the element of a that is specified by b.

vec_splats

Purpose

Returns a vector of which all the elements are set to a given value.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

`d=vec_splats(a)`

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 177. Result and argument types

d	a
vector signed char	signed char
vector unsigned char	unsigned char
vector signed short	signed short
vector unsigned short	unsigned short
vector signed int	signed int
vector unsigned int	unsigned int
vector signed long long	signed long long

Table 177. Result and argument types (continued)

d	a
vector unsigned long long	unsigned long long
vector float	float
vector double	double

Result value

Each element of the result has the value of a.

vec_splat_s8
Purpose

Returns a vector of which all the elements are set to a given value.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

`d=vec_splat_s8(a)`

Result and argument types

The following table describes the types of the returned value and the function argument.

Table 178. Types of the returned value and function argument

d	a[1]
vector signed char	signed int

Note:
1. a must be a signed literal with a value in the range -16 to 15 inclusive.

Result value

Each element of the result has the value of a.

vec_splat_s16
Purpose

Returns a vector of which all the elements are set to a given value.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

`d=vec_splat_s16(a)`

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 179. Types of the returned value and function arguments

d	a[1]
vector signed short	signed int

Note:

1. a must be a signed literal with a value in the range -16 to 15 inclusive.

Result value

Each element of the result has the value of a.

vec_splat_s32
Purpose

Returns a vector of which all the elements are set to a given value.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_splat_s32(a)
```

Result and argument types

The following table describes the types of the returned value and the function argument.

Table 180. Types of the returned value and function argument

d	a[1]
vector signed int	signed int

Note:

1. a must be a signed literal with a value in the range -16 to 15 inclusive.

Result value

Each element of the result has the value of a.

vec_splat_u8
Purpose

Returns a vector of which all the elements are set to a given value.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_splat_u8(a)
```

Result and argument types

The following table describes the types of the returned value and the function argument.

Table 181. Types of the returned value and function argument

d	a[1]
vector unsigned char	signed int

Note:

1. a must be a signed literal with a value in the range -16 to 15 inclusive.

Result value

The bit pattern of a is interpreted as an unsigned value. Each element of the result is given this value.

vec_splat_u16
Purpose

Returns a vector of which all the elements are set to a given value.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

`d=vec_splat_u16(a)`

Result and argument types

The following table describes the types of the returned value and the function argument.

Table 182. Types of the returned value and function argument

d	a[1]
vector unsigned short	signed int

Note:

1. a must be a signed literal with a value in the range -16 to 15 inclusive.

Result value

The bit pattern of a is interpreted as an unsigned value. Each element of the result is given this value.

vec_splat_u32
Purpose

Returns a vector of which all the elements are set to a given value.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

`d=vec_splat_u32(a)`

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 183. Types of the returned value and function arguments

d	a¹
vector unsigned int	signed int

Note:

1. a must be a signed literal with a value in the range -16 to 15 inclusive.

Result value

The bit pattern of a is interpreted as an unsigned value. Each element of the result is given this value.

vec_sqrt
Purpose

Returns a vector containing the square root of each element in the given vector.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_sqrt(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 184. Result and argument types

d	a
vector float	vector float
vector double	vector double

vec_sr
Purpose

Performs a right shift for each element of a vector.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_sr(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 185. Result and argument types

d	a	b
The same type as argument a	vector signed char	vector unsigned char
	vector unsigned char	vector unsigned char
	vector signed short	vector unsigned short
	vector unsigned short	vector unsigned short
	vector signed int	vector unsigned int
	vector unsigned int	vector unsigned int
	vector signed long long	vector unsigned long long
	vector unsigned long long	vector unsigned long long

Result value

Each element of the result vector is the result of right shifting the corresponding element of a by the number of bits specified by the value of the corresponding element of b, modulo the number of bits in the element. The bits that are shifted out are replaced by zeroes.

vec_sra
Purpose

Performs an algebraic right shift for each element of a vector.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

`d=vec_sra(a, b)`

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 186. Result and argument types

d	a	b
vector signed char	vector signed char	vector unsigned char
vector unsigned char	vector unsigned char	
vector signed short	vector signed short	vector unsigned short
vector unsigned short	vector unsigned short	
vector signed int	vector signed int	vector unsigned int
vector unsigned int	vector unsigned int	
vector signed long long	vector signed long long	vector unsigned long long
vector unsigned long long	vector unsigned long long	

Result value

Each element of the result vector is the result of algebraically right shifting the corresponding element of a by the number of bits specified by the value of the

corresponding element of b, modulo the number of bits in the element. The bits that are shifted out are replaced by copies of the most significant bit of the element of a.

vec_srl

Purpose

Right shifts a vector by a given number of bits.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

`d=vec_srl(a,b)`

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 187. Types of the returned value and function arguments

d	a	b[1]
The same type as argument a	vector bool char	Any of the following types:
	vector signed char	vector unsigned char
	vector unsigned char	vector unsigned short
	vector bool short	vector unsigned int
	vector signed short	
	vector unsigned short	
	vector bool int	
	vector signed int	
	vector unsigned int	
	vector pixel	

Note:

1. The least significant three bits of all byte elements in b must be the same.

Result value

The result is produced by shifting the contents of a right by the number of bits specified by the last three bits of the last element of b. The bits that are shifted out are replaced by zeroes.

vec_sro

Purpose

Right shifts a vector by a given number of bytes.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

`d=vec_sro(a,b)`

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 188. Types of the returned value and function arguments

d	a	b
The same type as argument a	vector signed char	Any of the following types:
	vector unsigned char	vector signed char
	vector signed short	vector unsigned char
	vector unsigned short	
	vector signed int	
	vector unsigned int	
	vector float	
	vector pixel	

Result value

The result is produced by shifting the contents of a right by the number of bytes specified by bits 121 through 124 of b. The bits that are shifted out are replaced by zeroes.

vec_srv
Purpose

Right-shifts the elements of a given vector by a given number of bits.

This built-in function is valid only when -qarch(-mcpu) is set to utilize the POWER9 technology and the altivec.h file is included.

Syntax

```
d=vec_srv(a, b)
```

Result and argument types

The result and arguments are all of the vector unsigned char type.

Result value

Suppose the following information. Element i of the result vector contains the same bit string as from bit $8-S_i$ to bit $15-S_i$ of X_i, inclusive.

- S_i ($0<=i<=15$) denotes the value in the three least-significant bits of element i of b.
- X_0 denotes the value of the halfword formed by concatenating a zero byte and element 0 of a.
- X_i ($1<=i<=15$) denotes the value of the halfword formed by concatenating elements i and $i+1$ of a.

vec_st

Purpose

Stores a vector to memory at the given address.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
vec_st(a, b, c)
```

Argument types

This function does not return a value. The value of b is added to the address that is specified by c, and the sum is truncated to a multiple of 16 bytes. The value of a is then stored into this memory address.

The following table describes the types of the function arguments.

Table 189. Data type of function returned value and arguments

a	b	c
vector unsigned int	int	unsigned long*
vector signed int		signed long*

Table 189. Data type of function returned value and arguments (continued)

a	b	c
vector unsigned char	long	vector unsigned char*
		unsigned char*
vector signed char		vector signed char*
		signed char*
vector bool char		vector bool char*
		unsigned char*
		signed char*
vector unsigned short		vector unsigned short*
		unsigned short*
vector signed short		vector signed short*
		signed short*
vector bool short		vector bool short*
		unsigned short*
		short*
vector pixel		vector pixel*
		unsigned short*
		short*
vector unsigned int		vector unsigned int*
		unsigned int*
vector signed int		vector signed int*
		signed int*
vector bool int		vector bool int*
		unsigned int*
		int*
vector float		vector float*
		float*

vec_ste
Purpose

Stores a vector element into memory at the given address.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

`vec_ste(a,b,c)`

Argument types

This function does not return a value. The following table describes the types of the function arguments.

Table 190. Types of the function arguments

a	b	c
vector bool char	Any integral type	signed char *
		unsigned char *
vector signed char		signed char *
vector unsigned char		unsigned char *
vector bool short		signed short *
		unsigned short *
vector signed short		signed short *
vector unsigned short		unsigned short *
vector bool int		signed int *
		unsigned int *
vector signed int		signed int *
vector unsigned int		unsigned int *
vector float		float *
vector pixel		signed short *
		unsigned short *

Result value

The effective address is the sum of c and the address specified by c, truncated to a multiple of the size in bytes of an element of the result vector. The value of the element of a at the byte offset that corresponds to the four least significant bits of the effective address is stored into memory at the effective address.

vec_stl
Purpose

Stores a vector into memory at the given address, and marks the data as Least Recently Used.

This built-in function is valid only when you include the `altivec.h` file.

Syntax
`vec_stl(a,b,c)`

Argument types

This function does not return a value. The following table describes the types of the function arguments.

Table 191. Types of the function arguments

a	b	c
vector bool char	Any integral type	signed char *
		unsigned char *
		vector bool char *
vector signed char		signed char *
		vector signed char *
vector unsigned char		unsigned char *
		vector unsigned char *
vector bool short		signed short *
		unsigned short *
		vector bool short *
vector signed short		signed short *
		vector signed short *
vector unsigned short		unsigned short *
		vector unsigned short *
vector bool int		signed int *
		unsigned int *
		vector bool int *
vector signed int		signed int *
		vector signed int *
vector unsigned int		unsigned int *
		vector unsigned int *
vector float		float *
		vector float *
vector pixel		signed short *
		unsigned short *
		vector pixel *

Result value

b is added to the address specified by c, and the sum is truncated to a multiple of 16 bytes. The value of a is then stored into this memory address. The data is marked as Least Recently Used.

vec_sub

Purpose

Returns a vector containing the result of subtracting each element of b from the corresponding element of a.

This function emulates the operation on long long vectors.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_sub(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 192. Result and argument types

d	a	b
The same type as argument a	vector signed char	The same type as argument a
	vector unsigned char	
	vector signed short	
	vector unsigned short	
	vector signed int	
	vector unsigned int	
	vector signed long long	
	vector unsigned long long	
	vector float	
	vector double	

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. The arithmetic is modular for integer vectors.

vec_sub_u128
Purpose

Subtracts unsigned quadword values.

The function operates on vectors as 128-bit unsigned integers.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_sub_u128(a, b)
```

Result and argument types

The type of d, a, and b must be `vector unsigned char`.

Result value

Returns low 128 bits of a - b.

vec_subc

Purpose

Returns a vector containing the borrows produced by subtracting each set of corresponding elements of the given vectors.

This built-in function is valid only when you include the altivec.h file.

Syntax

```
d=vec_subc(a, b)
```

Result and argument types

The type of d, a, and b must be vector unsigned int.

Result value

The value of each element of the result is the value of the borrow produced by subtracting the value of the corresponding element of b from the value of the corresponding element of a. The value is 0 if a borrow occurred, or 1 if no borrow occurred.

vec_subc_u128

Purpose

Returns the carry bit of the 128-bit subtraction of two quadword values.

The function operates on vectors as 128-bit unsigned integers.

This built-in function is valid only when you include the altivec.h file.

Syntax

```
d=vec_subc_u128(a, b)
```

Result and argument types

The type of d, a, and b must be vector unsigned char.

Result value

Returns the carry out of a - b.

vec_sube_u128

Purpose

Subtracts unsigned quadword values with carry bit from previous operation.

The function operates on vectors as 128-bit unsigned integers.

This built-in function is valid only when you include the altivec.h file.

Syntax

```
d=vec_sube_u128(a, b, c)
```

Result and argument types

The type of d, a, b, and c must be `vector unsigned char`.

Result value

Returns the low 128 bits of a - b - (c & 1).

vec_subec_u128
Purpose

Gets the carry bit of the 128-bit subtraction of two quadword values with carry bit from the previous operation.

The function operates on vectors as 128-bit unsigned integers.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_subec_u128(a, b, c)
```

Result and argument types

The type of d, a, b, and c must be `vector unsigned char`.

Result value

Returns the carry out of a - b - (c & 1).

vec_subs
Purpose

Returns a vector containing the saturated differences of each set of corresponding elements of the given vectors.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_subs(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 193. Types of the returned value and function arguments

d	a	b
vector signed char	vector bool char	vector signed char
	vector signed char	vector bool char
	vector signed char	vector signed char

Table 193. Types of the returned value and function arguments (continued)

d	a	b
vector unsigned char	vector bool char	vector unsigned char
	vector unsigned char	vector bool char
	vector unsigned char	vector unsigned char

Result value

The value of each element of the result is the saturated result of subtracting the value of the corresponding element of b from the value of the corresponding element of a.

vec_sum2s
Purpose

Returns a vector containing the results of performing a sum across 1/2 vector operation on two given vectors.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_sum2s(a, b)
```

Result and argument types

The type of d, a, and b must be `vector signed int`.

Result value

The first and third elements of the result are 0. The second element of the result contains the saturated sum of the first and second elements of a and the second element of b. The fourth element of the result contains the saturated sum of the third and fourth elements of a and the fourth element of b.

```
d[0] = 0
d[1] = a[0] + a[1] + b[1]
d[2] = 0
d[3] = a[2] + a[3] + b[3]
```

vec_sum4s
Purpose

Returns a vector containing the results of performing a sum across 1/4 vector operation on two given vectors.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_sum4s(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 194. Types of the returned value and function arguments

d	a	b
vector signed int	vector signed char	vector signed int
vector unsigned int	vector unsigned char	vector unsigned int
vector signed int	vector signed short	vector signed int

Result value

For each element n of the result vector, the value is obtained as follows:

- If a is of type `vector signed char` or `vector unsigned char`, the value is the saturated addition of elements 4n through 4n+3 of a and element n of b.

  ```
  d[0] = a[0]  + a[1]  + a[2]  + a[3]  + b[0]
  d[1] = a[4]  + a[5]  + a[6]  + a[7]  + b[1]
  d[2] = a[8]  + a[9]  + a[10] + a[11] + b[2]
  d[3] = a[12] + a[13] + a[14] + a[15] + b[3]
  ```

- If a is of type `vector signed short`, the value is the saturated addition of elements 2n through 2n+1 of a and element n of b.

  ```
  d[0] = a[0] + a[1] + b[0]
  d[1] = a[2] + a[3] + b[1]
  d[2] = a[4] + a[5] + b[2]
  d[3] = a[6] + a[7] + b[3]
  ```

vec_sums
Purpose

Returns a vector containing the results of performing a sum across vector operation on the given vectors.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_sums(a, b)
```

Result and argument types

The type of d, a, and b must be `vector signed int`.

Result value

The first three elements of the result are 0. The fourth element is the saturated sum of all the elements of a and the fourth element of b.

vec_test_data_class
Purpose

Determines the data class of the elements of the given vector.

This built-in function is valid only when `-qarch(-mcpu)` is set to utilize the POWER9 technology and the `altivec.h` file is included.

Syntax

```
d=vec_test_data_class(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 195. Types of the returned value and function arguments

d	a	b
vector bool int	vector float	unsigned char
vector bool long long	vector double	unsigned char

Result value

Returns the results of testing a for the condition selected by b. The value of b is in the range of 0-127. Each bit of b enables the test of a different condition. You can refer to the following table for the mapping relations between testing conditions and bits of b:

Table 196. Bitmask for testing conditions

Bits of b	Test conditions
0x01	Test for -Denormal
0x02	Test for +Denormal
0x04	Test for -Zero
0x08	Test for +Zero
0x10	Test for -Infinity
0x20	Test for +Infinity
0x40	Test for NaN

If any of the enabled test conditions is true, all of the bits of corresponding element are set to 1. If all of the enabled test conditions are false, all of the bits of the corresponding element is set to 0.

vec_trunc

Purpose

Returns a vector containing the truncated values of the corresponding elements of the given vector.

Note: `vec_trunc` is another name for `vec_roundz`. For details, see "vec_roundz" on page 468.

This built-in function is valid only when you include the `altivec.h` file.

vec_unpackh
Purpose

Unpacks the most significant half of a vector into another vector.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_unpackh(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 197. Result and argument types

d	a
vector signed short	vector signed char
vector signed int	vector signed short
vector signed long long	vector signed int
vector bool long long	vector bool int

Result value

The value of each element of the result is the value of the corresponding element of the most significant half of a.

Related reference:

"-maltivec (-qaltivec)" on page 129

Related information:

Vector element order toggling

vec_unpackl
Purpose

Unpacks the least significant half of a vector into another vector.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_unpackl(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 198. Result and argument types

d	a
vector signed short	vector signed char
vector signed int	vector signed short
vector signed long long	vector signed int
vector bool long long	vector bool int

Result value

The value of each element of the result is the value of the corresponding element of the least significant half of a.

Related reference:

"-maltivec (-qaltivec)" on page 129

Related information:

Vector element order toggling

vec_vclz
Purpose

Computes the count of leading zero bits of each element of the given vector.

This built-in function is valid only when you include the altivec.h file.

Syntax

d=vec_vclz(a)

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a
vector unsigned char	vector unsigned char
vector signed char	vector signed char
vector unsigned short	vector unsigned short
vector signed short	vector signed short
vector unsigned int	vector unsigned int
vector signed int	vector signed int
vector unsigned long long	vector unsigned long long
vector signed long long	vector signed long long

Result value

Each element of the result is set to the number of leading zeros of the corresponding element of a.

Related reference:

"vec_cntlz" on page 405

vec_vgbbd
Purpose

Performs a gather-bits-by-bytes operation on the given vector.

This built-in function is valid only when you include the altivec.h file.

Syntax

`d=vec_vgbbd(a)`

Result and argument types

The following table describes the types of the returned value and the function arguments.

d	a
vector unsigned char	vector unsigned char
vector signed char	vector signed char

Result value

Each doubleword element of the result is set as follows:

Let x(i) (0 <= i < 8) denote the byte elements of the corresponding input doubleword element, with x(7) as the most significant byte. For each pair of i and j (0 <= i < 8, 0 <= j < 8), the jth bit of the ith byte element of the result is set to the value of the ith bit of the jth byte element of the input.

Related reference:

"vec_gbb" on page 425

vec_xl

Purpose

Loads a 16-byte vector from the memory address specified by the displacement a and the pointer b.

Note: It is preferred that you use vector pointers and the indirection operator * instead of this function to load vectors.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

`d=vec_xl(a, b)`

Result and argument types

The following table describes the types of the function returned value and the function arguments.

Table 199. Data type of function returned value and arguments

d	a	b
vector signed char	long	signed char *
		const signed char *
		vector signed char *
		const vector signed char *
vector unsigned char		unsigned char *
		const unsigned char *
		vector unsigned char *
		const vector unsigned char *
vector signed short		signed short *
		const signed short *
		vector signed short *
		const vector signed short *
vector unsigned short		unsigned short *
		const unsigned short *
		vector unsigned short *
		const vector unsigned short *
vector signed int		signed int *
		const signed int *
		vector signed int *
		const vector signed int *
vector unsigned int		unsigned int *
		const unsigned int *
		vector unsigned int *
		const vector unsigned int *
vector signed long long		signed long long *
		const signed long long *
		vector signed long long *
		const vector signed long long *
vector unsigned long long		unsigned long long *
		const unsigned long long *
		vector unsigned long long *
		const vector unsigned long long *
vector float		float *
		const float *
		vector float *
		const vector float *
vector double		double *
		const double *
		vector double *
		const vector double *

Result value

vec_xl adds the displacement provided by a to the address provided by b to obtain the effective address for the load operation. It does not truncate the effective address to a multiple of 16 bytes.

The order of elements in the function result is big endian when **-maltivec=be** (**-qaltivec=be**) is in effect. Otherwise, the order is little endian.

vec_xl_be
Purpose

Loads a 16-byte vector from the memory address specified by the displacement a and the pointer b.

Note: It is preferred that you use vector pointers and the indirection operator * instead of this function to load vectors.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_xl_be(a, b)
```

Result and argument types

The following table describes the types of the function returned value and the function arguments.

Table 200. Data type of function returned value and arguments

d	a	b
vector signed char	long	signed char *
		const signed char *
		vector signed char *
		const vector signed char *
vector unsigned char		unsigned char *
		const unsigned char *
		vector unsigned char *
		const vector unsigned char *
vector signed short		signed short *
		const signed short *
		vector signed short *
		const vector signed short *
vector unsigned short		unsigned short *
		const unsigned short *
		vector unsigned short *
		const vector unsigned short *
vector signed int		signed int *
		const signed int *
		vector signed int *
		const vector signed int *
vector unsigned int		unsigned int *
		const unsigned int *
		vector unsigned int *
		const vector unsigned int *
vector signed long long		signed long long *
		const signed long long *
		vector signed long long *
		const vector signed long long *
vector unsigned long long		unsigned long long *
		const unsigned long long *
		vector unsigned long long *
		const vector unsigned long long *
vector float		float *
		const float *
		vector float *
		const vector float *
vector double		double *
		const double *
		vector double *
		const vector double *

Result value

vec_xl_be adds the displacement provided by a to the address provided by b to obtain the effective address for the load operation. It does not truncate the effective address to a multiple of 16 bytes.

The order of elements in the function result is big endian regardless of the -maltivec (-qaltivec) option in effect.

vec_xl_len
Purpose

Returns a vector that loads a given number of bytes from the given address.

This built-in function is valid only when -qarch(-mcpu) is set to utilize the POWER9 technology and the altivec.h file is included.

Syntax

d=vec_xl_len(a, b)

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 201. Types of the returned value and function arguments

d	a	b
vector signed char	signed char *	size_t
vector unsigned char	unsigned char *	size_t
vector signed short	signed short *	size_t
vector unsigned short	unsigned short *	size_t
vector signed int	signed int *	size_t
vector unsigned int	unsigned int *	size_t
vector signed long long	signed long long *	size_t
vector unsigned long long	unsigned long long *	size_t
vector float	float *	size_t
vector double	double *	size_t

Result value

The result is loaded from the memory address that is specified by a. The number of bytes loaded is specified by b. Bytes of elements are initialized in order from the byte stream which is defined by the endianness of the operating environment. Any byte of elements that is not initialized is set to a 0 value.

The behavior is undefined if the value of b is outside of the range 0 - 255.

vec_xl_len_r

Purpose

Loads a string of bytes into vector register, right-justified. Sets the leftmost elements (16-*cnt*) to 0.

This built-in function is valid only when `-qarch(-mcpu)` is set to utilize the POWER9 technology and the `altivec.h` file is included.

Prototype

vector unsigned char vec_xl_len_r (unsigned char *ptr*, const int *cnt*);

Parameters

ptr
   Points to a base address.

cnt
   The number of bytes to load. The value of *cnt* must be in the range 1 - 16.

vec_xst_len_r

Purpose

Stores a right-justified string of bytes.

This built-in function is valid only when `-qarch(-mcpu)` is set to utilize the POWER9 technology and the `altivec.h` file is included.

Prototype

void vec_xst_len_r (vector unsigned char *data*, unsigned char *ptr*, const int *cnt*);

Parameters

data
   Address displacement.

ptr
   Points to a base address.

cnt
   The number of bytes to store. The value of *cnt* must be in the range 1 - 16 and must be a compile-time known constant.

vec_xld2

Purpose

Loads a 16-byte vector from two 8-byte elements at the memory address specified by the displacement a and the pointer b.

Note: It is preferred that you use vector pointers and the indirection operator `*` instead of this function to load vectors.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_xld2(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 202. Result and argument types

d	a	b
vector signed char	long	signed char *
vector unsigned char		unsigned char *
vector signed short		signed short *
vector unsigned short		unsigned short *
vector signed int		signed int *
vector unsigned int		unsigned int *
vector signed long long		signed long long *
vector unsigned long long		unsigned long long *
vector float		float *
vector double		double *

Result value

This function adds the displacement and the pointer R-value to obtain the address for the load operation. It does not truncate the effective address to a multiple of 16 bytes.

Related reference:

"-maltivec (-qaltivec)" on page 129

Related information:

Vector element order toggling

vec_xlds
Purpose

Loads an 8-byte element from the memory address specified by the displacement a and the pointer b and then splats it onto a vector.

Note: It is preferred that you use vector pointers and the indirection operator * instead of this function to load vectors.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_xlds(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 203. Result and argument types

d	a	b
vector signed long long	long	signed long long *
vector unsigned long long	long	unsigned long long *
vector double	long	double *

Result value

This function adds the displacement and the pointer R-value to obtain the address for the load operation. It does not truncate the effective address to a multiple of 16 bytes.

vec_xlw4

Purpose

Loads a 16-byte vector from four 4-byte elements at the memory address specified by the displacement a and the pointer b.

Note: It is preferred that you use vector pointers and the indirection operator * instead of this function to load vectors.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_xlw4(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 204. Result and argument types

d	a	b
vector signed char	long	signed char *
vector unsigned char		unsigned char *
vector signed short		signed short *
vector unsigned short		unsigned short *
vector signed int		signed int *
vector unsigned int		unsigned int *
vector float		float *

Result value

This function adds the displacement and the pointer R-value to obtain the address for the load operation. It does not truncate the effective address to a multiple of 16 bytes.

Related reference:

"-maltivec (-qaltivec)" on page 129

Related information:

Vector element order toggling

vec_xor

Purpose

Performs a bitwise XOR of the given vectors.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

`d=vec_xor(a, b)`

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 205. Result and argument types

d	a	b
vector bool char	vector bool char	vector bool char
vector signed char	vector bool char	vector signed char
	vector signed char	vector signed char
		vector bool char
vector unsigned char	vector bool char	vector unsigned char
	vector unsigned char	vector unsigned char
		vector bool char
vector bool short	vector bool short	vector vector bool short
vector signed short	vector bool short	vector signed short
	vector signed short	vector signed short
		vector bool short
vector unsigned short	vector bool short	vector unsigned short
	vector unsigned short	vector unsigned short
		vector bool short
vector bool int	vector bool int	vector bool int
vector signed int	vector bool int	vector signed int
	vector signed int	vector signed int
		vector bool int
vector unsigned int	vector bool int	vector unsigned int
	vector unsigned int	vector unsigned int
		vector bool int
vector bool long long	vector bool long long	vector bool long long
vector signed long long	vector bool long long	vector signed long long
	vector signed long long	vector signed long long
		vector bool long long

Table 205. Result and argument types (continued)

d	a	b
vector unsigned long long	vector bool long long	vector unsigned long long
	vector unsigned long long	vector unsigned long long
		vector bool long long
vector float	vector bool int	vector float
	vector float	vector bool int
		vector float
vector double	vector bool long long	vector double
	vector double	vector bool long long
		vector double

Result value

The result is the bitwise XOR of a and b.

vec_xst_len
Purpose

Stores a given byte length of a vector to a given address.

This built-in function is valid only when -qarch(-mcpu) is set to utilize the POWER9 technology and the altivec.h file is included.

Syntax

```
void vec_xst_len(a, b, c)
```

Argument types

The following table describes the types of the function arguments.

Table 206. Types of function arguments

a	b	c
vector signed char	signed char *	size_t
vector unsigned char	unsigned char *	size_t
vector signed short	signed short *	size_t
vector unsigned short	unsigned short *	size_t
vector signed int	signed int *	size_t
vector unsigned int	unsigned int *	size_t
vector signed long long	signed long long *	size_t
vector unsigned long long	unsigned long long *	size_t
vector float	float *	size_t
vector double	double *	size_t

Result

No value is returned. The specified bytes of a are stored into an address. The number of bytes to be stored is specified by c and the address is specified by b.

The behavior is undefined if the value of c is outside of 0-255. The behavior is implementation defined if the value of c is not a multiple of the vector element size.

vec_xst

Purpose

Stores the elements of the 16-byte vector a to the effective address obtained by adding the displacement provided by b with the address provided by c. The effective address is not truncated to a multiple of 16 bytes.

Note: It is preferred that you use vector pointers and the indirection operator * instead of this function to store vectors.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_xst(a, b, c)
```

Result and argument types

The following table describes the types of the function returned value and the function arguments.

Table 207. Types of the returned value and the function arguments

d	a	b	c
void	vector signed char	long	signed char *
			vector signed char *
	vector unsigned char		unsigned char *
			vector unsigned char *
	vector signed short		signed short *
			vector signed short *
	vector unsigned short		unsigned short *
			vector unsigned short *
	vector signed int		signed int *
			vector signed int *
	vector unsigned int		unsigned int *
			vector unsigned int *
	vector signed long long		signed long long *
			vector signed long long *
	vector unsigned long long		unsigned long long *
			vector unsigned long long *
	vector float		float *
			vector float *
	vector double		double *
			vector double *

vec_xst_be

Purpose

Stores the elements of the 16-byte vector a in big endian element order to the effective address obtained by adding the displacement provided by b with the address provided by c. The effective address is not truncated to a multiple of 16 bytes.

Note: It is preferred that you use vector pointers and the indirection operator * instead of this function to store vectors.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_xst_be(a, b, c)
```

Result and argument types

The following table describes the types of the function returned value and the function arguments.

Table 208. Types of the returned value and the function arguments

d	a	b	c
void	vector signed char	long	signed char *
			vector signed char *
	vector unsigned char		unsigned char *
			vector unsigned char *
	vector signed short		signed short *
			vector signed short *
	vector unsigned short		unsigned short *
			vector unsigned short *
	vector signed int		signed int *
			vector signed int *
	vector unsigned int		unsigned int *
			vector unsigned int *
	vector signed long long		signed long long *
			vector signed long long *
	vector unsigned long long		unsigned long long *
			vector unsigned long long *
	vector float		float *
			vector float *
	vector double		double *
			vector double *

vec_xstd2

Purpose

Puts a 16-byte vector a as two 8-byte elements to the memory address specified by the displacement b and the pointer c.

Note: It is preferred that you use vector pointers and the indirection operator * instead of this function to store vectors.

This built-in function is valid only when you include the `altivec.h` file.

Syntax

```
d=vec_xstd2(a, b, c)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 209. Result and argument types

d	a	b	c
void	vector signed char	long	signed char *
	vector unsigned char		unsigned char *
	vector signed short		signed short *
	vector unsigned short		unsigned short *
	vector signed int		signed int *
	vector unsigned int		unsigned int *
	vector signed long long		signed long long *
	vector unsigned long long		unsigned long long *
	vector float		float *
	vector double		double *
	vector pixel		signed short * or unsigned short *

Result value

This function adds the displacement and the pointer R-value to obtain the address for the store operation. It does not truncate the effective address to a multiple of 16 bytes.

Related reference:

"-maltivec (-qaltivec)" on page 129

Related information:

Vector element order toggling

vec_xstw4
Purpose

Puts a 16-byte vector a to four 4-byte elements at the memory address specified by the displacement b and the pointer c.

Note: It is preferred that you use vector pointers and the indirection operator *
instead of this function to store vectors.

This built-in function is valid only when you include the altivec.h file.

Syntax

```
d=vec_xstw4(a, b, c)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 210. Result and argument types

d	a	b	c
void	vector signed char	long	signed char *
	vector unsigned char		unsigned char *
	vector signed short		signed short *
	vector unsigned short		unsigned short *
	vector signed int		signed int *
	vector unsigned int		unsigned int *
	vector float		float *
	vector pixel		signed short * or unsigned short *

Result value

This function adds the displacement and the pointer R-value to obtain the address for the store operation. It does not truncate the effective address to a multiple of 16 bytes.

Related reference:

"-maltivec (-qaltivec)" on page 129

Related information:

Vector element order toggling

GCC atomic memory access built-in functions (IBM extension)

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 211. 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 212. 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_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_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_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.

__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.

Atomic operation and fetch functions

__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_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_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.

__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.

Atomic compare and swap functions

__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`.

__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.

GCC object size checking built-in functions

IBM XL C/C++ for Linux, V13.1.6 supports object size checking built-in functions that are provided by GCC. With these functions, you can detect and prevent some buffer overflow attacks.

The GCC object size checking built-in functions are grouped into the following categories.

Related information:

↪ Object size checking built-in functions in GCC documentation

__builtin_object_size
Purpose

When used with **-O2** or higher optimization, returns a constant number of bytes from the given pointer to the end of the object pointed to if the size of object is known at compile time.

Prototype

size_t __builtin_object_size (void *ptr*, int *type*);

Parameters

ptr
The pointer of the object.

type
An integer constant that is in the range 0 - 3 inclusive. If the pointer points to multiple objects at compile time, *type* determines whether this function returns the maximum or minimum of the remaining byte counts in those objects. If the object that a pointer points to is enclosed in another object, *type* determines whether the whole variable or the closest surrounding subobject is considered to be the object that the pointer points to.

Return value

Table 213 describes the return values of this built-in function when both of the following conditions are met.

- **-O2** or higher optimization level is in effect.
- The objects that *ptr* points to can be determined at compile time.

If any of these conditions are not met, this built-in function returns the values as described in Table 214 on page 522.

Table 213. Return values when both conditions are met

type	Return value
0	The maximum of the sizes of all objects. The whole variable is considered to be the object that *ptr* points to.
1	The maximum of the sizes of all objects. The closest surrounding variable is considered to be the object that *ptr* points to.

Table 213. Return values when both conditions are met (continued)

type	Return value
2	The minimum of the sizes of all objects. The whole variable is considered to be the object that *ptr* points to.
3	The minimum of the sizes of all objects. The closest surrounding variable is considered to be the object that *ptr* points to.

Table 214. Return values when any conditions are not met

type	Return value
0	(size_t) -1
1	(size_t) -1
2	(size_t) 0
3	(size_t) 0

Note: IBM XL C/C++ for Linux, V13.1.6 does not support the multiple targets and closest surrounding features. You can assign a value in the range 0 - 3 to *type*, but the compiler behavior is as if *type* were 0.

Examples

Consider the file `myprogram.c`:

```
#include "stdio.h"

int func(char *a){
  char b[10];
  char *p = &b[5];
  printf("__builtin_object_size(a,0):%ld\n",__builtin_object_size(a,0));
  printf("__builtin_object_size(b,0):%ld\n",__builtin_object_size(b,0));
  printf("__builtin_object_size(p,0):%ld\n",__builtin_object_size(p,0));
  return 0;
}

int main(){
  char a[10];
  func(a);
  return 0;
}
```

- If you compile `myprogram.c` with the **-0** option, you get the following output:

```
__builtin_object_size(a,0):10
__builtin_object_size(b,0):10
__builtin_object_size(p,0):5
```

- If you compile `myprogram.c` with the **-0** and **-qnoinline** options, you get the following output:

```
__builtin_object_size(a,0):-1
/* The objects the pointer points to cannot be determined at compile time. */
__builtin_object_size(b,0):10
__builtin_object_size(p,0):5
```

__builtin___*_chk

In addition to __builtin_object_size, IBM XL C/C++ for Linux, V13.1.6 also supports *_chk* built-in functions for some common string operation functions; for example, __builtin___memcpy_chk is provided for `memcpy`. When these built-in

functions are used with **-02** or higher optimization level, the compiler issues a warning message if it can determine at compile time that the object will always be overflowed; the built-in functions are optimized to the corresponding string functions such as memcpy when either of the following conditions is met:

- The last argument of these functions is (size_t) -1.
- It is known at compile time that the destination object will not be overflowed.

The supported built-in functions for common string operation functions are described in the following table.

Table 215. Checking built-in functions for string operation functions

Function	Built-in function	Prototype
memcpy	__builtin___memcpy_chk	void * __builtin___memcpy_chk (void *dest, const void *src, size_t n, size_t os);
mempcpy	__builtin___mempcpy_chk	void * __builtin___mempcpy_chk (void *dest, const void *src, size_t n, size_t os);
memmove	__builtin___memmove_chk	void * __builtin___memmove_chk (void *dest, const void *src, size_t n, size_t os);
memset	__builtin___memset_chk	void * __builtin___memset_chk (void *s, int c, size_t n, size_t os);
strcpy	__builtin___strcpy_chk	char * __builtin___strcpy_chk (char *dest, const char *src, size_t os);
strncpy	__builtin___strncpy_chk	char * __builtin___strncpy_chk (char *dest, const char *src, size_t n, size_t os);
stpcpy	__builtin___stpcpy_chk	char * __builtin___stpcpy_chk (char *dest, const char *src, size_t os);
strcat	__builtin___strcat_chk	char * __builtin___strcat_chk (char *dest, const char *src, size_t os);
strncat	__builtin___strncat_chk	char * __builtin___strncat_chk (char *dest, const char *src, size_t n, size_t os);

There are other checking built-in functions as described in the following table. The corresponding library functions are called when you use these built-in functions.

Table 216. Other checking built-in functions

Function	Built-in function	Prototype
sprintf	__builtin___sprintf_chk	int __builtin___sprintf_chk (char *s, int flag, size_t os, const char *fmt, ...);

Table 216. Other checking built-in functions (continued)

Function	Built-in function	Prototype
snprintf	__builtin___snprintf_chk	int __builtin___snprintf_chk (char *s, size_t maxlen, int flag, size_t os);
vsprintf	__builtin___vsprintf_chk	int __builtin___vsprintf_chk (char *s, int flag, size_t os, const char *fmt,va_list ap);
vsnprintf	__builtin___vsnprintf_chk	int __builtin___vsnprintf_chk (char *s, size_t maxlen, int flag, size_t os, const char *fmt, va_list ap);
printf	__builtin___printf_chk	int __builtin___printf (int flag, const char *format, ...);
vprintf	__builtin___vprintf_chk	int __builtin___vprintf (int flag, const char *format, va_list ap);
fprintf	__builtin___fprintf_chk	int __builtin___fprintf (FILE *stream, int flag, const char *format, ...);
vfprintf	__builtin___vfprintf_chk	int __builtin___vfprintf (FILE *stream, int flag, const char *format, va_list ap);

Note: In the prototype of each function, the ellipsis (...) represents an optional list of parameters. IBM XL C/C++ for Linux ignores these optional parameters and protects all globally accessible variables.

Miscellaneous built-in functions

Miscellaneous functions are grouped into the following categories:

- "Optimization-related functions"
- "Move to/from register functions" on page 525
- "Memory-related functions" on page 527

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

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.

__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 the 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
> The value to be set to the special purpose register. It 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. In addition, when you run **-qprefetch=assistthread**, the compiler uses the delinquent load information to perform analysis and generate prefetching assist threads. For more information, see "-qprefetch" on page 184.

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;
}
```

Related information

- "-qprefetch" on page 184

Transactional memory built-in functions

Transactional memory is a model for parallel programming. This module provides functions that allow you to designate a block of instructions or statements to be treated atomically. Such an atomic block is called a transaction. When a thread executes a transaction, all of the memory operations within the transaction occur simultaneously from the perspective of other threads.

For some kinds of parallel programs, a transaction implementation can be more efficient than other implementation methods, such as locks. You can use these built-in functions to mark the beginning and end of transactions, and to diagnose the reasons for failure.

In the transactional memory built-in functions, the *TM_buff* parameter allows for a user-provided memory location to be used to store the transaction state and debugging information.

The transactional state is entered following a successful call to __TM_begin or __TM_simple_begin, and ended by __TM_end, __TM_abort, __TM_named_abort, or by transaction failure. Alternate code is needed to handle situations where transaction failures are persistent.

Note: Debugging your application while it is inside a transactional state might result in unpredictable behavior.

Transaction failure occurs when any of the following conditions is met:

- Memory that is accessed in the transactional state is accessed by another thread or by the same thread running in the suspended state before the transaction completes.
- The architecture-defined footprint for memory accesses within a transaction is exceeded.
- The architecture-defined nesting limit for nested transactions is exceeded.

Transactions can be nested. You can use `__TM_begin` or `__TM_simple_begin` in the transactional state. Within an outermost transaction initiated with `__TM_begin`, nested transactions must be initiated with `__TM_simple_begin`, or by `__TM_begin` using the same buffer of the outermost containing transaction.

A nested transaction is subsumed into the containing transaction. Therefore, a failure of the nested transaction is treated as a failure of all containing transactions, and the nested transaction completes only when all contained transactions complete.

Note: You must include the `htmxlintrin.h` file in the source code if you use any of the transactional memory built-in functions.

Transaction begin and end functions

__TM_begin
Purpose

Marks the beginning of a transaction.

Prototype

long __TM_begin (void* const *TM_buff*);

Parameter

TM_buff
The address of a 16-byte transaction diagnostic block (TDB) that contains diagnostic information.

Usage

Upon a transaction failure (including a user abort), execution resumes from the point immediately following the `__TM_begin` that initiated the failed transaction as if the `__TM_begin` were unsuccessful. The diagnostic information is transferred from the TEXASR and TFIAR registers to *TM_buff*.

You can use the transaction inquiry functions to query the transaction status.

Return value

This function returns _HTM_TBEGIN_STARTED if successful; otherwise, it returns a different value.

Related information

- "__TM_simple_begin"
- "Transaction inquiry functions" on page 532

__TM_end
Purpose

Marks the end of a transaction.

Prototype

long __TM_end ();

Return value

The return value is _HTM_TBEGIN_STARTED if the thread is in the transactional state before the instruction starts; otherwise, it returns a different value.

__TM_simple_begin
Purpose

Marks the beginning of a transaction.

Prototype

long __TM_simple_begin ();

Usage

Upon a transaction failure (including a user abort), execution resumes from the point immediately following the __TM_simple_begin function that initiated the failed transaction as if the __TM_simple_begin were unsuccessful. The diagnostic information is saved in the TEXASR register.

The transaction status of transactions started using __TM_simple_begin cannot be queried by using the transaction inquiry functions.

Return value

This function returns _HTM_TBEGIN_STARTED if successful; otherwise, it returns a different value.

Related information

- "__TM_begin" on page 530
- "Transaction inquiry functions" on page 532

Transaction abort functions

__TM_abort
Purpose

Aborts a transaction with failure code 0.

Prototype

> void __TM_abort ();

Related information

- "__TM_named_abort"

__TM_named_abort
Purpose

Aborts a transaction with the specified failure code.

Prototype

void __TM_named_abort (unsigned char const *code*);

Parameter

code
> The specified failure code. It is a literal that is in the range of 0 - 255.

Related information

- "__TM_abort" on page 531

Transaction inquiry functions

__TM_failure_address
Purpose

Gets the code address at which the most recent transaction was aborted.

Prototypes

long __TM_failure_address (void* const *TM_buff*);

Parameter

TM_buff
> The address of a 16-byte transaction diagnostic block (TDB) that contains diagnostic information.

Return value

This function returns the address at which the most recent transaction was aborted. The address is obtained from the TFIAR register.

__TM_failure_code
Purpose

Provides the raw failure code for the transaction.

Prototypes

long long __TM_failure_code (void* const *TM_buff*);

Parameter

TM_buff
    The address of a 16-byte transaction diagnostic block (TDB) that contains
    diagnostic information.

Return value

The function returns the raw failure code for the transaction. The raw failure code
is obtained from the TEXASR register.

__TM_is_conflict
Purpose

Queries whether the transaction was aborted because of a conflict.

Prototypes

long __TM_is_conflict (void* const *TM_buff*);

Parameter

TM_buff
    The address of a 16-byte transaction diagnostic block (TDB) that contains
    diagnostic information.

Return value

This function returns 1 if both of the following qualifications are met; otherwise, it
returns 0:
- The TDB is valid.
- The transaction was aborted because of a conflict. Bit 11, 12, 13, and 14 of the
  TEXASR register are ORed as 1.

__TM_is_failure_persistent
Purpose

Queries whether the transaction was aborted because of a persistent reason.

Prototypes

long __TM_is_failure_persistent (void* const *TM_buff*);

Parameter

TM_buff
    The address of a 16-byte transaction diagnostic block (TDB) that contains
    diagnostic information.

Return value

This function returns 1 if the transaction was aborted because of a persistent
reason; bit 7 of the TEXASR register is 1. Otherwise, the function returns 0.

__TM_is_footprint_exceeded
Purpose

Queries whether the transaction was aborted because of exceeding the maximum number of cache lines.

Prototypes

long __TM_is_footprint_exceeded (void* const *TM_buff*);

Parameter

TM_buff
> The address of a 16-byte transaction diagnostic block (TDB) that contains diagnostic information.

Return value

This function returns 1 if both of the following qualifications are met; otherwise, it returns 0:

* The TDB is valid.
* The transaction was aborted because the maximum number of cache lines was exceeded. Bit 10 of the TEXASR register is 1.

__TM_is_illegal
Purpose

Queries whether the transaction was aborted because of an illegal attempt, such as an instruction not permitted in transactional mode or other kind of illegal access.

Prototypes

long __TM_is_illegal (void* const *TM_buff*);

Parameter

TM_buff
> The address of a 16-byte transaction diagnostic block (TDB) that contains diagnostic information.

Return value

This function returns 1 if both of the following qualifications are met; otherwise, it returns 0:

* The TDB is valid.
* The transaction was aborted because of an illegal attempt. Bit 8 of the TEXASR register is 1.

__TM_is_named_user_abort
Purpose

Queries whether the transaction failed because of a user abort instruction and gets the transaction abort code.

Prototypes

long __TM_is_named_user_abort (void* const *TM_buff*, unsigned char* *code*);

Parameter

code
> The address of the memory location to save the transaction abort code.

TM_buff
> The address of a 16-byte transaction diagnostic block (TDB) that contains diagnostic information.

Return value

This function returns 1 if both of the following qualifications are met; otherwise, it returns 0:

- The TDB is valid.
- The transaction failed because of a user abort instruction. Bit 31 of the TEXASR register is 1.

When both of the preceding qualifications are met, *code* is set to bit 0 - 7 of the TEXASR register. The value of *code* is also passed to the tabort hardware instruction. When either of the preceding qualifications is not met, *code* is set to 0.

Related information

- "__TM_is_user_abort"

__TM_is_nested_too_deep
Purpose

Queries whether the transaction was aborted because of trying to exceed the maximum nesting depth.

Prototypes

long __TM_is_nested_too_deep (void* const *TM_buff*);

Parameter

TM_buff
> The address of a 16-byte transaction diagnostic block (TDB) that contains diagnostic information.

Return value

This function returns 1 if both of the following qualifications are met; otherwise, it returns 0:

- The TDB is valid.
- The transaction was aborted because of trying to exceed the maximum nesting depth. Bit 9 of the TEXASR register is 1.

__TM_is_user_abort
Purpose

Queries whether the transaction failed because of a user abort instruction.

Prototypes

long __TM_is_user_abort (void* const *TM_buff*);

Parameter

TM_buff
　　The address of a 16-byte transaction diagnostic block (TDB) that contains
　　diagnostic information.

Return value

This function returns 1 if both of the following qualifications are met; otherwise, it
returns 0:

• The TDB is valid.

• The transaction failed because of a user abort instruction. Bit 31 of the TEXASR
　register is 1.

Related information

• "__TM_is_named_user_abort" on page 534

__TM_nesting_depth
Purpose

Returns the current nesting depth. If the thread is not in the transactional state, the
function returns the depth at which the most recent transaction was aborted.

Prototypes

long __TM_nesting_depth (void* const *TM_buff*);

Parameter

TM_buff
　　The address of a 16-byte transaction diagnostic block (TDB) that contains
　　diagnostic information.

Return value

If the thread is in the transactional state, this function returns the current nesting
depth. Otherwise, the function returns the depth at which the most recent
transaction was aborted. The function returns 0 if the transaction is completed
successfully.

The current nesting depth is obtained from bit 52 - 63 of the TEXASR register.

Transaction resume and suspend functions

__TM_resume
Purpose

Resumes a transaction.

Prototype

void __TM_resume ();

__TM_suspend
Purpose

Suspends a transaction.

Prototype

void __TM_suspend ();

Supported GCC vector built-in functions

The following GCC vector built-in functions are also supported in IBM XL C/C++ for Linux, V13.1.6.

- vec_vsx_ld
- vec_vsx_st
- vec_xxsldi
- vec_dstt
- vec_xxpermdi

The following tables describe how vector built-in functions in GCC and IBM XL C/C++ for Linux, V13.1.6 are mapped.

Table 217. Vector built-in function mappings: vec_vsx_ld and vec_xl

Supported GCC vector built-in function: vec_vsx_ld	Equivalent IBM XL C/C++ for Linux function: vec_xl
vector double vec_vsx_ld (int, const vector double *)	vector double vec_xl (signed long, const vector double *)
vector double vec_vsx_ld (int, const double *)	vector double vec_xl (signed long, const double *)
vector float vec_vsx_ld (int, const vector float *)	vector float vec_xl (signed long, const vector float *)
vector float vec_vsx_ld (int, const float *)	vector float vec_xl (signed long, const float *)
vector bool int vec_vsx_ld (int, const vector bool int *)	
vector signed int vec_vsx_ld (int, const vector signed int *)	vector signed int vec_xl (signed long, const vector signed int *)
vector signed int vec_vsx_ld (int, const int *)	vector signed int vec_xl (signed long, const signed int *)
vector signed int vec_vsx_ld (int, const long *)	
vector unsigned int vec_vsx_ld (int, const vector unsigned int *)	vector unsigned int vec_xl (signed long, const vector unsigned int *)
vector unsigned int vec_vsx_ld (int, const unsigned int *)	vector unsigned int vec_xl (signed long, const unsigned int *)
vector unsigned int vec_vsx_ld (int, const unsigned long *)	
vector bool short vec_vsx_ld (int, const vector bool short *)	

Table 217. Vector built-in function mappings: vec_vsx_ld and vec_xl (continued)

Supported GCC vector built-in function: vec_vsx_ld	Equivalent IBM XL C/C++ for Linux function: vec_xl
vector pixel vec_vsx_ld (int, const vector pixel *)	
vector signed short vec_vsx_ld (int, const vector signed short *)	vector signed short vec_xl (signed long, const vector signed short *)
vector signed short vec_vsx_ld (int, const short *)	vector signed short vec_xl (signed long, const signed short *)
vector unsigned short vec_vsx_ld (int, const vector unsigned short *)	vector unsigned short vec_xl (signed long, const vector unsigned short *)
vector unsigned short vec_vsx_ld (int, const unsigned short *)	vector unsigned short vec_xl (signed long, const unsigned short *)
vector bool char vec_vsx_ld (int, const vector bool char *)	
vector signed char vec_vsx_ld (int, const vector signed char *)	vector signed char vec_xl (signed long, const vector signed char *)
vector signed char vec_vsx_ld (int, const signed char *)	vector signed char vec_xl (signed long, const signed char *)
vector unsigned char vec_vsx_ld (int, const vector unsigned char *)	vector unsigned char vec_xl (signed long, const vector unsigned char *)
vector unsigned char vec_vsx_ld (int, const unsigned char *)	vector unsigned char vec_xl (signed long, const unsigned char *)

Table 218. Vector built-in function mappings: vec_vsx_st and vec_xst

Supported GCC vector built-in function: vec_vsx_st	Equivalent IBM XL C/C++ for Linux function: vec_xst
void vec_vsx_st (vector double, int, vector double *)	void vec_xst (vector double, signed long, const vector double *)
void vec_vsx_st (vector double, int, double *)	void vec_xst (vector double, signed long, double *)
void vec_vsx_st (vector float, int, vector float *)	void vec_xst (vector float, signed long, const vector float *)
void vec_vsx_st (vector float, int, float *)	void vec_xst (vector float, signed long, float *)
void vec_vsx_st (vector signed int, int, vector signed int *)	void vec_xst (vector signed int, signed long, const vector signed int *)
void vec_vsx_st (vector signed int, int, int *)	void vec_xst (vector signed int, signed long, signed int *)
void vec_vsx_st (vector unsigned int, int, vector unsigned int *)	void vec_xst (vector unsigned short, signed long, const vector unsigned short *)
void vec_vsx_st (vector unsigned int, int, unsigned int *)	void vec_xst (vector unsigned int, signed long, unsigned int*)
void vec_vsx_st (vector bool int, int, vector bool int *)	

Table 218. Vector built-in function mappings: vec_vsx_st and vec_xst (continued)

Supported GCC vector built-in function: vec_vsx_st	Equivalent IBM XL C/C++ for Linux function: vec_xst
void vec_vsx_st (vector bool int, int, unsigned int *)	
void vec_vsx_st (vector bool int, int, int *)	
void vec_vsx_st (vector signed short, int, vector signed short *)	void vec_xst (vector signed short, signed long, const vector signed short *)
void vec_vsx_st (vector signed short, int, short *)	void vec_xst (vector signed short, signed long, signed short *)
void vec_vsx_st (vector unsigned short, int, vector unsigned short *)	void vec_xst (vector unsigned short, signed long, const vector unsigned short *)
void vec_vsx_st (vector unsigned short, int, unsigned short *)	void vec_xst (vector unsigned short, signed long, unsigned short *)
void vec_vsx_st (vector bool short, int, vector bool short *)	
void vec_vsx_st (vector bool short, int, unsigned short *)	
void vec_vsx_st (vector pixel, int, vector pixel *)	
void vec_vsx_st (vector pixel, int, unsigned short *)	
void vec_vsx_st (vector pixel, int, short *)	
void vec_vsx_st (vector bool short, int, short *)	
void vec_vsx_st (vector signed char, int, vector signed char *)	void vec_xst (vector signed char, signed long, const vector signed char *)
void vec_vsx_st (vector signed char, int, signed char *)	void vec_xst (vector signed char, signed long, signed char *)
void vec_vsx_st (vector unsigned char, int, vector unsigned char *)	void vec_xst (vector unsigned char, signed long, const vector unsigned char *)
void vec_vsx_st (vector unsigned char, int, unsigned char *)	void vec_xst (vector unsigned char, signed long, unsigned char *)
void vec_vsx_st (vector bool char, int, vector bool char *)	
void vec_vsx_st (vector bool char, int, unsigned char *)	
void vec_vsx_st (vector bool char, int, signed char *)	

Table 219. Vector built-in function mappings: vec_xxsldi and vec_sldw

Supported GCC vector built-in function: vec_xxsldi	Equivalent IBM XL C/C++ for Linux function: vec_sldw
vector double vec_xxsldi (vector double, vector double, int)	vector double vec_sldw (vector double, vector double, signed int)

Table 219. Vector built-in function mappings: vec_xxsldi and vec_sldw (continued)

Supported GCC vector built-in function: vec_xxsldi	Equivalent IBM XL C/C++ for Linux function: vec_sldw
vector float vec_xxsldi (vector float, vector float, int)	
vector long long vec_xxsldi (vector long long, vector long long, int)	vector signed long long vec_sldw (vector signed long long, vector signed long long, signed int)
vector unsigned long long vec_xxsldi (vector unsigned long long, vector unsigned long long, int)	vector unsigned long long vec_sldw (vector unsigned long long, vector unsigned long long, signed int)
vector int vec_xxsldi (vector int, vector int, int)	vector signed int vec_sldw (vector signed int, vector signed int, signed int)
vector unsigned int vec_xxsldi (vector unsigned int, vector unsigned int, int)	vector unsigned int vec_sldw (vector unsigned int, vector unsigned int, signed int)
vector short vec_xxsldi (vector short, vector short, int)	vector signed short vec_sldw (vector signed short, vector signed short, signed int)
vector unsigned short vec_xxsldi (vector unsigned short, vector unsigned short, int)	vector unsigned short vec_sldw (vector unsigned short, vector unsigned short, signed int)
vector signed char vec_xxsldi (vector signed char, vector signed char, int)	vector signed char vec_sldw (vector signed char, vector signed char, signed int)
vector unsigned char vec_xxsldi (vector unsigned char, vector unsigned char, int)	vector unsigned char vec_sldw (vector unsigned char, vector unsigned char, signed int)

Table 220. Vector built-in function mappings: vec_dstt and vec_dstt

Supported GCC vector built-in function: vec_dstt	Equivalent IBM XL C/C++ for Linux function: vec_dstt
void vec_dstt (const unsigned long *, signed int, signed int)	
void vec_dstt (const signed long *, signed int, signed int)	

The vec_permi built-in function in the following table maps to the vec_xxpermdi function only when -maltivec=be (-qaltivec=be) is set.

Table 221. Vector built-in function mappings: vec_xxpermdi and vec_permi

Supported GCC vector built-in function: vec_xxpermdi	Equivalent IBM XL C/C++ for Linux function: vec_permi
vector double vec_xxpermdi (vector double, vector double, int)	vector double vec_permi(vector double, vector double, signed int)
vector long long vec_xxpermdi (vector long long, vector long long, int)	vector signed long long vec_permi(vector signed long long, vector signed long long, signed int)

Supported GCC vector built-in function: vec_xxpermdi	Equivalent IBM XL C/C++ for Linux function: vec_permi
vector unsigned long long vec_xxpermdi (vector unsigned long long, vector unsigned long long, int)	vector unsigned long long vec_permi(vector unsigned long long, vector unsigned long long, signed int)
int vec_xxpermdi (vector int, vector int, int)	vector signed long long vec_permi(vector signed long long, vector signed long long, signed int)
unsigned int vec_xxpermdi (vector unsigned int, vector unsigned int, int)	vector unsigned long long vec_permi(vector unsigned long long, vector unsigned long long, signed int)
vector short vec_xxpermdi (vector short, vector short, int)	vector signed long long vec_permi(vector signed long long, vector signed long long, signed int)
vector unsigned short vec_xxpermdi (vector unsigned short, vector unsigned short, int)	vector unsigned long long vec_permi(vector unsigned long long, vector unsigned long long, signed int)
vector signed char vec_xxpermdi (vector signed char, vector signed char, int)	vector signed long long vec_permi(vector signed long long, vector signed long long, signed int)
vector unsigned char vec_xxpermdi (vector unsigned char, vector unsigned char, int)	vector unsigned long long vec_permi(vector unsigned long long, vector unsigned long long, signed int)

Related information

- vec_xl
- vec_xst
- vec_sldw
- vec_dstt
- vec_permi

Supported GCC non-vector built-in functions

IBM XL C/C++ for Linux, V13.1.6 supports the following GCC non-vector built-in functions:

- __atomic_add
- __builtin___fprintf_chk
- __builtin___memccpy_chk
- __builtin___memcpy_chk
- __builtin___memmove_chk
- __builtin___mempcpy_chk
- __builtin___memset_chk
- __builtin___printf_chk
- __builtin___snprintf_chk
- __builtin___sprintf_chk
- __builtin___stpcpy_chk
- __builtin___stpncpy_chk
- __builtin___strcat_chk

- __builtin___strcpy_chk
- __builtin___strlcat_chk
- __builtin___strlcpy_chk
- __builtin___strncat_chk
- __builtin___strncpy_chk
- __builtin___vfprintf_chk
- __builtin___vprintf_chk
- __builtin___vsnprintf_chk
- __builtin___vsprintf_chk
- __builtin_abort
- __builtin_abs
- __builtin_acos
- __builtin_acosf
- __builtin_acosh
- __builtin_acoshf
- __builtin_acoshl
- __builtin_acosl
- __builtin_alloca
- __builtin_asin
- __builtin_asinf
- __builtin_asinh
- __builtin_asinhf
- __builtin_asinhl
- __builtin_asinl
- __builtin_atan2
- __builtin_atan2f
- __builtin_atan2l
- __builtin_atan
- __builtin_atanf
- __builtin_atanh
- __builtin_atanhf
- __builtin_atanhl
- __builtin_atanl
- __builtin_bcmp
- __builtin_bcopy
- __builtin_bswap16
- __builtin_bswap32
- __builtin_bswap64
- __builtin_bzero
- __builtin_cabs
- __builtin_cabsf
- __builtin_cabsl
- __builtin_cacos
- __builtin_cacosf
- __builtin_cacosh

- __builtin_cacoshf
- __builtin_cacoshl
- __builtin_cacosl
- __builtin_carg
- __builtin_cargf
- __builtin_cargl
- __builtin_casin
- __builtin_casinf
- __builtin_casinh
- __builtin_casinhf
- __builtin_casinhl
- __builtin_casinl
- __builtin_catan
- __builtin_catanf
- __builtin_catanh
- __builtin_catanhf
- __builtin_catanhl
- __builtin_catanl
- __builtin_cbrt
- __builtin_cbrtf
- __builtin_cbrtl
- __builtin_ccos
- __builtin_ccosf
- __builtin_ccosh
- __builtin_ccoshf
- __builtin_ccoshl
- __builtin_ccosl
- __builtin_ceil
- __builtin_ceilf
- __builtin_ceill
- __builtin_ceill
- __builtin_cexp
- __builtin_cexpl
- __builtin_cimag
- __builtin_cimagf
- __builtin_cimagl
- __builtin_clog
- __builtin_clogf
- __builtin_clogl
- __builtin_clz
- __builtin_clzl
- __builtin_clzll
- __builtin_conj
- __builtin_conjf
- __builtin_conjl

- __builtin_copysign
- __builtin_copysignf
- __builtin_copysignl
- __builtin_cos
- __builtin_cosf
- __builtin_cosh
- __builtin_coshf
- __builtin_coshl
- __builtin_cosl
- __builtin_cpow
- __builtin_cpowf
- __builtin_cpowl
- __builtin_cproj
- __builtin_cprojf
- __builtin_cprojl
- __builtin_creal
- __builtin_crealf
- __builtin_creall
- __builtin_csin
- __builtin_csinf
- __builtin_csinh
- __builtin_csinhf
- __builtin_csinhl
- __builtin_csinl
- __builtin_csqrt
- __builtin_csqrtf
- __builtin_csqrtl
- __builtin_ctan
- __builtin_ctanf
- __builtin_ctanh
- __builtin_ctanhf
- __builtin_ctanhl
- __builtin_ctanl
- __builtin_ctz
- __builtin_ctzl
- __builtin_ctzll
- __builtin_darn
- __builtin_darn_32
- __builtin_darn_raw
- __builtin_erf
- __builtin_erfc
- __builtin_erfcf
- __builtin_erfcl
- __builtin_erff
- __builtin_erfl

- __builtin_exp2
- __builtin_exp2f
- __builtin_exp2l
- __builtin_exp
- __builtin_expect
- __builtin_expf
- __builtin_expl
- __builtin_expm1
- __builtin_expm1f
- __builtin_expm1l
- __builtin_fabs
- __builtin_fabsf
- __builtin_fabsl
- __builtin_fabss
- __builtin_fdimf
- __builtin_fdiml
- __builtin_ffs
- __builtin_ffsl
- __builtin_ffsll
- __builtin_floor
- __builtin_floorf
- __builtin_floorl
- __builtin_fma
- __builtin_fmaf
- __builtin_fmal
- __builtin_fmax
- __builtin_fmaxf
- __builtin_fmaxl
- __builtin_fmin
- __builtin_fminf
- __builtin_fminl
- __builtin_fmod
- __builtin_fmodf
- __builtin_fmodl
- __builtin_fpclassify
- __builtin_fprintf
- __builtin_fprintf
- __builtin_frexp
- __builtin_frexpf
- __builtin_frexpl
- __builtin_huge_val
- __builtin_huge_valf
- __builtin_huge_vall
- __builtin_hypot
- __builtin_hypotf

- __builtin_hypotl
- __builtin_ilogb
- __builtin_ilogbf
- __builtin_ilogbl
- __builtin_index
- __builtin_isfinite
- __builtin_isgreater
- __builtin_isgreaterequal
- __builtin_isgreaterequall
- __builtin_isgreaterl
- __builtin_isinf
- __builtin_isless
- __builtin_islessequal
- __builtin_islessequall
- __builtin_islessgreater
- __builtin_islessgreaterl
- __builtin_islessl
- __builtin_isnan
- __builtin_isnormal
- __builtin_isunordered
- __builtin_isunorderedl
- __builtin_labs
- __builtin_ldexp
- __builtin_ldexpf
- __builtin_ldexpl
- __builtin_lgamma
- __builtin_lgammaf
- __builtin_lgammal
- __builtin_llabs
- __builtin_llrint
- __builtin_llrintf
- __builtin_llrintl
- __builtin_llround
- __builtin_llroundf
- __builtin_llroundl
- __builtin_log10
- __builtin_log10f
- __builtin_log10l
- __builtin_log1p
- __builtin_log1pf
- __builtin_log1pl
- __builtin_log2
- __builtin_log2f
- __builtin_log2l
- __builtin_log

- __builtin_logb
- __builtin_logbf
- __builtin_logbl
- __builtin_logf
- __builtin_logl
- __builtin_longjmp
- __builtin_lrint
- __builtin_lrintf
- __builtin_lrintl
- __builtin_lround
- __builtin_lroundf
- __builtin_lroundl
- __builtin_memchr
- __builtin_memcmp
- __builtin_memcpy
- __builtin_memmove
- __builtin_mempcpy
- __builtin_memset
- __builtin_modf
- __builtin_modff
- __builtin_modfl
- __builtin_nan
- __builtin_nanf
- __builtin_nanl
- __builtin_nans
- __builtin_nansf
- __builtin_nansl
- __builtin_nearbyint
- __builtin_nearbyintf
- __builtin_nearbyintl
- __builtin_nextafterf
- __builtin_nextafterl
- __builtin_nexttoward
- __builtin_nexttowardf
- __builtin_nexttowardl
- __builtin_nop
- __builtin_object_size
- __builtin_parity
- __builtin_parityl
- __builtin_parityll
- __builtin_popcount
- __builtin_popcountl
- __builtin_popcountll
- __builtin_pow
- __builtin_powf

- __builtin_powi
- __builtin_powif
- __builtin_powil
- __builtin_powl
- __builtin_prefetch
- __builtin_printf
- __builtin_remainderf
- __builtin_remainderl
- __builtin_remquof
- __builtin_remquol
- __builtin_rindex
- __builtin_rint
- __builtin_rintf
- __builtin_rintl
- __builtin_round
- __builtin_roundf
- __builtin_roundl
- __builtin_scalbln
- __builtin_scalblnf
- __builtin_scalblnl
- __builtin_scalbn
- __builtin_scalbnf
- __builtin_scalbnl
- __builtin_setjmp
- __builtin_signbit
- __builtin_signbitf
- __builtin_signbitl
- __builtin_sin
- __builtin_sincosf
- __builtin_sinf
- __builtin_sinh
- __builtin_sinhf
- __builtin_sinhl
- __builtin_sinl
- __builtin_sprintf
- __builtin_sqrt
- __builtin_sqrtf
- __builtin_sqrtfs
- __builtin_sqrtl
- __builtin_stpcpy
- __builtin_strcat
- __builtin_strchr
- __builtin_strcmp
- __builtin_strcpy
- __builtin_strcspn

- __builtin_strlen
- __builtin_strncat
- __builtin_strncmp
- __builtin_strncpy
- __builtin_strpbrk
- __builtin_strrchr
- __builtin_strspn
- __builtin_strstr
- __builtin_tan
- __builtin_tanf
- __builtin_tanh
- __builtin_tanhf
- __builtin_tanhl
- __builtin_tanint
- __builtin_tanl
- __builtin_tgamma
- __builtin_tgammaf
- __builtin_tgammal
- __builtin_trap
- __builtin_trunc
- __builtin_truncf
- __builtin_truncl
- __builtin_unreachable
- __builtin_vsnprintf
- __builtin_vsprintf
- __exchange_and_add

Chapter 9. OpenMP runtime functions for parallel processing

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 19

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*);

Parameter

lock
> Must be a variable of type `omp_lock_t` that is initialized with **omp_init_lock** or **omp_init_nest_lock**.

omp_get_active_level

Purpose

Returns the number of nested, active parallel regions enclosing the task that contains the call. The routine always returns a non-negative integer, and returns 0 if it is called from the sequential part of the program.

Prototype

int omp_get_active_level(void);

omp_get_ancestor_thread_num

Purpose

Returns the thread number of the ancestor of the current thread at a given nested level. 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*);

Parameter

level
> Specifies a given nested level of the current thread.

omp_get_default_device

Purpose

The **omp_get_default_device** function returns the default target device.

Prototype

> int omp_get_default_device(void);

Usage

The function returns the value of the *default-device-var* ICV of the current task.

▶ GPU ◀ When the function is called from within a target region on a target device, the effect is unspecified. GPU ◀

omp_get_dynamic

Purpose

Returns non-zero if dynamic thread adjustment is enabled and returns 0 otherwise.

Prototype

> int omp_get_dynamic (void);

omp_get_initial_device

Purpose

The **omp_get_initial_device** function returns the device number of the host device.

Prototype

> int omp_get_initial_device(void);

Usage

The value of the device number is implementation defined. If it is in the range of 0 and one less than **omp_get_num_devices**(), it can be used with all device constructs and functions; if it is outside that range, it can be used with the device memory functions but not in the **device** clause.

▶ GPU ◀ When the function is called from within a target region on a target device, the effect is unspecified. GPU ◀

omp_get_level

Purpose

Returns the number of active and inactive nested parallel regions that the generating task is executing in. This does not include the implicit parallel region. Returns 0 if it is called from the sequential part of the program. Otherwise, returns a non-negative integer.

Prototype

int omp_get_level(void);

omp_get_max_active_levels

Purpose

Returns 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** runtime routine.

Prototype

int omp_get_max_active_levels(void);

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_nested

Purpose

Returns non-zero if nested parallelism is enabled and 0 if it is disabled.

Prototype

int omp_get_nested (void);

omp_get_num_devices

Purpose

The **omp_get_num_devices** function returns the number of target devices.

Prototype

> int omp_get_num_devices(void);

Usage

The function returns the number of available target devices.

▶ `GPU` When the function is called from within a target region on a target device, the effect is unspecified. `GPU` ◀

omp_get_num_places

Purpose

Returns the number of places that are available to the execution environment in the place list. This value is equivalent to the number of places in the *place-partition-var* internal control variable (ICV) in the execution environment of the initial task.

Prototype

> int omp_get_num_places(void);

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_get_num_teams

Purpose

The **omp_get_num_teams** function returns the number of teams in the current teams region.

Prototype

> int omp_get_num_teams (void);

Usage

When the function is called from outside of a teams region, it returns 1.

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_get_partition_place_nums

Purpose

Returns the list of place numbers that correspond to the places in the *place-partition-var* internal control variable (ICV) of the innermost implicit task. The *place-partition-var* ICV controls the place partition that is available to the execution environment for encountered parallel regions. Each implicit task has one copy of the *place-partition-var* ICV.

Prototype

void omp_get_partition_place_nums(int *place_nums*);

Parameter

place_nums
An integer array that contains places in the place partition of the innermost implicit task.

Usage

The size of the array *place_nums* that contains place numbers must be equal to or larger than the return value of **omp_get_partition_num_places()**; otherwise, the behavior is undefined.

omp_get_partition_num_places

Purpose

Returns the number of places in the place partition of the innermost implicit task.

Prototype

int omp_get_partition_num_places(void);

omp_get_place_num

Purpose

Returns the place number of the place to which the encountering thread is bound.

Prototype

int omp_get_place_num(void);

Usage

When the encountering thread is bound to a place, the function returns the place number that is associated with the thread. The returned value is between -1 and the return value of **omp_get_num_places()** exclusive. When the encountering thread is not bound to a place, the function returns -1.

omp_get_place_num_procs

Purpose

Returns the number of processors that are available to the execution environment in the specified place.

Prototype

int omp_get_place_num_procs(int *place_num*);

Parameter

place_num
A positive integer that represents the number of the place.

Usage

The function returns the number of processors that are associated with the place whose number is *place_num*. The function returns zero when *place_num* is negative or is equal to or larger than the result value of **omp_get_num_places()**.

omp_get_place_proc_ids

Purpose

Returns the numerical identifiers of the processors that are available to the execution environment in the specified place.

Prototype

void omp_get_place_proc_ids(int *place_num*, int **ids*);

Parameter

place_num
A positive integer that represents the number of a place.

ids
An integer array.

Usage

The function returns the non-negative numerical identifiers of each processor that is associated with the place that is numbered *place_num*. The numerical identifiers are returned in the array *ids* whose size must be equal to or larger than the return value of **omp_get_place_num_procs()**; otherwise, the behavior is undefined. The function has no effect when *place_num* is a negative value or is equal to or larger than the return value of **omp_get_num_places()**.

omp_get_proc_bind

Purpose

Returns the thread affinity policy to be applied for the subsequent nested parallel regions that do not specify a **proc_bind** clause. The thread affinity policy can be one of the following values as defined in omp.h:

- `omp_proc_bind_false`
- `omp_proc_bind_true`
- `omp_proc_bind_master`
- `omp_proc_bind_close`
- `omp_proc_bind_spread`

Prototype

omp_proc_bind_t omp_get_proc_bind(void);

Related information:

"OMP_PROC_BIND" on page 32

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.

Note: The affinity schedule type has been deprecated and might be removed in a future release. You can use the dynamic schedule type for a similar functionality.

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" on page 563

Related information:

"OMP_SCHEDULE" on page 36

omp_get_team_num

Purpose

The **omp_get_team_num** function returns the team number of the calling thread.

Prototype

int omp_get_team_num (void);

Usage

The team number is an integer in the range of 0 and one less than the number of teams in the current teams region, inclusive.

When the function is called from outside of a teams region, it returns zero.

Related reference:

"omp_get_num_teams" on page 554

omp_get_team_size

Purpose

Returns the thread team size that the ancestor or the current thread 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*);

Parameter

level
    Specifies a given nested level of the current thread.

omp_get_thread_limit

Purpose

Returns the maximum number of OpenMP threads available to the program. The value is stored in the *thread-limit-var* internal control variable. *thread-limit-var* can be set with the *OMP_THREAD_LIMIT* environment variable.

Prototype

int omp_get_thread_limit(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_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.

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_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_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*);

Parameter

lock
>Must be a variable of type `omp_lock_t`.

omp_is_initial_device

Purpose

The **omp_is_initial_device** function tests whether the current task is executing on the host device.

Prototype

int omp_is_initial_device(void);

Result Value

If the current task is executing on the host device, the function returns nonzero. Otherwise, it returns zero.

omp_set_default_device

Purpose

The **omp_set_default_device** function controls the default target device by setting the *default-device-var* internal control variable (ICV).

Prototype

void omp_set_default_device(int *device_num*);

Parameter

device_num
>Must be the device number of the host device or be non-negative and less than the number of target devices.

Usage

` GPU ` When the function is called from within a target region on a target device, the effect of it is unspecified. ` GPU `

Related reference:

"omp_get_initial_device" on page 552

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*);

Parameter

dynamic_threads
Indicates whether the number of threads available in subsequent parallel region can be adjusted by the runtime library. If *dynamic_threads* is nonzero, the runtime library can adjust the number of threads. If *dynamic_threads* is zero, the runtime library cannot dynamically adjust the number of threads.

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*);

Parameter

lock
Must be a variable of type omp_lock_t that is initialized with omp_init_lock or omp_init_nest_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_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 levels of parallelism, the value of *max-active-levels-var* is set to the number of parallel levels supported by the run time. If the number of parallel levels requested is not a positive integer, this routine call is ignored.

When nested parallelism is turned off, this routine has no effect 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.

Use **omp_set_max_active_levels** only in serial regions of a program. This routine
has no effect in parallel regions of a program.

Prototype

void omp_set_max_active_levels(int *max_levels*);

Parameter

max_levels
  An integer that specifies the maximum number of nested, active parallel
  regions.

omp_set_nested

Purpose

Enables or disables nested parallelism.

Prototype

void omp_set_nested (int *nested*);

Usage

If the argument to **omp_set_nested** evaluates to true, nested parallelism is enabled
for the current task; otherwise, nested parallelism is disabled for the current task.
The setting of **omp_set_nested** overrides the setting of the *OMP_NESTED*
environment variable.

Note: If the number of threads in a parallel region and its nested parallel regions
exceeds the number of available processors, your program might suffer
performance degradation.

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*);

Parameter

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_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, or static, *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:

Related information:

omp_target_alloc

Purpose

The **omp_target_alloc** function allocates memory in a device data environment.

Prototype

void* omp_target_alloc(size_t *size*, int *device_num*);

Parameters

size
> Specifies the size in bytes of the storage memory to be allocated.

device_num
> Must be the device number of the host device or be non-negative and less than the number of target devices.

Usage

The device address returned by this function can be used in a **is_device_ptr** clause on the **omp target** directive.

Pointer arithmetic is not supported on the device address that is returned by this function.

You can free the storage that is returned by this function only by using the **omp_target_free** function; otherwise, the compiler behavior is unspecified.

If it cannot allocate the memory in the device data environment dynamically, it returns NULL.

▶ `GPU` You cannot call this function on a target device. `GPU` ◀

Related reference:

"#pragma omp target" on page 279

"omp_get_initial_device" on page 552

"omp_get_num_devices" on page 553

"omp_target_free" on page 565

omp_target_associate_ptr

Purpose

The **omp_target_associate_ptr** function associates a device pointer with a host pointer. When the host pointer appears in a subsequent **map** clause, the associated device pointer is used as the target for data motion that is associated with that host pointer.

Prototype

> int omp_target_associate_ptr(void * *host_ptr*, void * *device_ptr*, size_t *size*, size_t *device_offset*, int *device_num*);

Parameters

host_ptr
Specifies the address in the memory of the host device.

device_ptr
Specifies the address in the memory of the target device denoted by *device_num*.

size
Specifies the size in bytes of the buffer that is being pointed to.

device_offset
Specifies the offset to be applied to *device_ptr*. *device_ptr* with the offset *device_offset* is used as the base address for the device side of the mapping.

device_num
Must be the device number of the host device or be non-negative and less than the number of target devices.

Usage

The function returns zero on success and nonzero on failure.

You can associate only one device buffer with a given host pointer value and device number pair.

The result of associating pointers that share underlying storage is unspecified. You can use the **omp_target_is_present** function to test whether a given host pointer has a corresponding variable in the device data environment.

▶ `GPU` You cannot call this function on a target device. `GPU` ◀

Related reference:

"omp_get_initial_device" on page 552

"omp_get_num_devices" on page 553

"omp_target_is_present" on page 566

omp_target_disassociate_ptr

Purpose

The **omp_target_disassociate_ptr** function removes the associated pointer for a given device from a host pointer.

Prototype

> int omp_target_disassociate_ptr(void * *ptr*, int *device_num*);

Parameters

ptr
  Specifies the address in the memory of the host device.

device_num
  Must be the device number of the host device or be non-negative and less than the number of target devices.

Usage

After a call to this function, the contents of the device buffer are invalidated.

The result of calling this function on a pointer that is not NULL and does not have associated data on the given device is unspecified.

▶ `GPU` You cannot call this function on a target device. `GPU` ◀

Related reference:

"omp_get_initial_device" on page 552

"omp_get_num_devices" on page 553

omp_target_free

Purpose

The **omp_target_free** function frees the device memory allocated by the **omp_target_alloc** function.

Prototype

> void omp_target_free(void * *device_ptr*, int *device_num*);

Parameters

device_ptr
> Specifies the device address of a storage location that is returned by the **omp_target_alloc** function.

device_num
> Must be the device number of the host device or be non-negative and less than the number of target devices.

Usage

If *device_ptr* is NULL, the operation is ignored.

You must insert synchronization to ensure that all accesses to *device_ptr* are completed before the call to **omp_target_free**.

> **GPU** You cannot call this function on a target device. **GPU**

Related reference:

omp_target_is_present

Purpose

The **omp_target_is_present** function tests whether a host pointer has corresponding storage on a device.

Prototype

> int omp_target_is_present(void * *ptr*, int *device_num*);

Parameter

ptr
> Specifies the address in the memory of the host device.

device_num
> Must be the device number of the host device or be non-negative and less than the number of target devices.

Usage

This function returns nonzero if *ptr* is present on the device that is denoted by *device_num* by a **map** clause; otherwise, it returns zero.

> **GPU** You cannot call this function on a target device. **GPU**

omp_target_memcpy

Purpose

The **omp_target_memcpy** function copies memory between pointers, which can be either host or target device pointers.

Prototype

```
int omp_target_memcpy(void * dst, void * src,
        size_t length, size_t dst_offset, size_t src_offset,
        int dst_device_num, int src_device_num);
```

Parameters

dst
> Specifies the address in the memory of the destination device.

src
> Specifies the address in the memory of the source device.

length
> Specifies the number of bytes of memory to be copied.

dst_offset
> Specifies the offset to be applied to *dst*.

src_offset
> Specifies the offset to be applied to *src*.

dst_device_num
> Must be the device number of the host device or be non-negative and less than
> the number of target devices. It represents the destination device of the copy.

src_device_num
> Must be the device number of the host device or be non-negative and less than
> the number of target devices. It represents the source device of the copy.

Usage

The *length* bytes of memory at offset *src_offset* from *src* in the device data
environment of device *src_device_num* are copied to *dst* starting at offset *dst_offset* in
the device data environment of device *dst_device_num*.

It returns zero on success and nonzero on failure.

▶ GPU You cannot call this function on a target device. GPU ◀

Related reference:
"omp_get_initial_device" on page 552
"omp_get_num_devices" on page 553

---

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);
```

Parameter

lock
> Must be a variable of type `omp_lock_t` that is initialized with **omp_init_lock** or **omp_init_nest_lock**.

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*);

Parameter

lock
> Must be a variable of type `omp_lock_t` that is initialized with **omp_init_lock** or **omp_init_nest_lock**.

Notices

Programming interfaces: Intended programming interfaces allow the customer to write programs to obtain the services of IBM XL C/C++ for Linux.

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, MD-NC119
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:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, 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 want 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:

Intellectual Property Dept. for Rational Software
IBM Corporation
5 Technology Park Drive
Westford, MA   01886
U.S.A.

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. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

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, 2017.

PRIVACY POLICY CONSIDERATIONS:

IBM Software products, including software as a service solutions, ("Software Offerings") may use cookies or other technologies to collect product usage information, to help improve the end user experience, or to tailor interactions with the end user, or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering's use of cookies is set forth below.

This Software Offering does not use cookies or other technologies to collect personally identifiable information.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM's Privacy Policy at http://www.ibm.com/privacy and IBM's Online Privacy Statement at http://www.ibm.com/privacy/details in the section entitled "Cookies, Web Beacons and Other Technologies," and the "IBM Software Products and Software-as-a-Service Privacy Statement" at http://www.ibm.com/software/info/product-privacy.

Trademarks

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 is a registered trademark 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.

NVIDIA is either registered trademark or trademark of NVIDIA Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Index

A

alias
   pragma disjoint   241
alignment
   pragma pack   246
architecture
   macros   310

B

built-in functions
   atomic   350
   binary floating-point
      absolute value   327
      conversion   329
      FPSCR   332
      multiply-add/substract   334
      reciprocal estimate   335
      rounding   335
      selection   337
      software division   338
      square root   337
   binary-coded decimal
      addition   340
      comparison   341
      load   348
      overflow testing   349
      storage   348
      substraction   340
   binary-coded decimal (BCD)   340
   block   365
   cache   357
   cryptography   360
   fixed point
      absolute value   315
      assert   316
      bit permutation   317
      division   320
      load   321
      multiply   322
      overview   315
      population counting   323
      rotation   324
      storage   326
      trap   326
      zero counting   319
   floating-point   327
   GCC atomic memory access
      compare and swap   520
      fetch and operation   514
      lock   513
      operation and fetch   517
      overview   512
      release   513
      synchronization   513
   miscellaneous functions   524
   object size checking   521
   synchronization   350
   transactional memory   529
   vectors   365

C

cache-related built-in functions
   data   357
   prefetch   359
combination
   constructs   297
compiler commands
   cleanpdf   299
   genhtml   299
   mergepdf   300
   showpdf   301
compiler generated information
   listing   13
   overview   11
   paging space errors   15
compiler options
   performance optimization   60
   resolving conflicts   7
   specifying compiler options
      command lines   5
      configuration files   6
      source files   6
   summary   51
compiler options category
   customization   64
   floating-point   54
   input control   52
   language element control   53
   object code control   55
   output control   51
   portability and migration   64
   template control   54
compiler predefined macros
   overview   305
compiler reference
   configuring
      defaults   17
   OpenMP runtime functions   551
   options
      individual options   65
      individual pragmas   240
   options overview   51
   pragma directive syntax   239
   pragmas   239
      scope   239
configuration
   custom configuration files   38
configuration file attributes
   overview   43
configuration files
   code attributes   43
   customizing   43
   editing   43
   ipa attributes   44
   options attributes   44
cryptography built-in functions
   Hash   362
   miiscellaneous functions   363
   standard function   360

D

defaults
   customizing   43

E

edits
   configuration files   43
environment variables
   compile time   18
   link time   18
   OpenMP
      default device   25
      display environment   25
      dynamic adjustment   27
      nested active parallel regions   28
      nested parallelism   28
      overview   25
      parallel regions threads   29
      places list   30
      schedule types   36
      stack size   36
      thread affinity   32
      waiting threads   38
      whole program threads   37
   runtime
      options setting   20
      overview   18
   scheduling
      algorithm   36
   setting   17
   XLSMPOPTS   19
error checking
   runtime check   141
error debugging
   all debugging information   150
error debugging
   line number debug   169

F

Fortran error messages
   return codes   12

G

GCC
   options
      summary   233

I

invocations
   compilers   1
   input files   3
   output files   4
   preprocessor   7
   selecting   1
   syntax   2

IBM ®

Product Number: 5765-J08; 5725-C73

Printed in USA