IBM XL C/C++ for Linux, V13.1.3

Compiler Reference for Little Endian Distributions

Version 13.1.3

SC27-6570-02

IBM XL C/C++ for Linux, V13.1.3

Compiler Reference for Little Endian Distributions

Version 13.1.3

SC27-6570-02

Note
Before using this information and the product it supports, read the information in "Notices" on page 467.

First edition

This edition applies to IBM XL C/C++ for Linux, V13.1.3 (Program 5765-J08; 5725-C73) and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

© Copyright IBM Corporation 1996, 2015.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

About this document ix
Who should read this document ix
How to use this document ix
How this document is organized ix
Conventions x
Related information xiii
IBM XL C/C++ information xiii
Standards and specifications xiv
Other IBM information xv
Other information xv
Technical support xv
How to send your comments xv

Chapter 1. Compiling and linking applications 1
Invoking the compiler 1
Command-line syntax 2
Types of input files 3
Types of output files 4
Specifying compiler options 5
Specifying compiler options on the command line 5
Specifying compiler options in a configuration file 5
Specifying compiler options in program source files 6
Resolving conflicting compiler options 6
Preprocessing 7
Directory search sequence for included files . . . 8
Linking 9
Order of linking 10
Redistributable libraries 11
Compiler messages and listings 11
Compiler messages 11
Compiler return codes 12
Compiler listings 13
Paging space errors during compilation 15

Chapter 2. Configuring compiler defaults 17
Setting environment variables 17
Compile-time and link-time environment variables 18
Runtime environment variables 18
Environment variables for parallel processing . . 19
Using custom compiler configuration files 37
Creating custom configuration files 38
Using IBM XL C/C++ for Linux, V13.1.3 with the Advance Toolchain 41
Editing the default configuration file 42
Configuration file attributes 42

Chapter 3. Tracking compiler license usage 45
Understanding compiler license tracking 45
Setting up SLM Tags logging 45

Chapter 4. Compiler options reference 47
Summary of compiler options by functional category 47
Output control 47
Input control 48
Language element control 49
Template control (C++ only) 50
Floating-point and integer control 50
Object code control 51
Error checking and debugging 52
Listings, messages, and compiler information . . 55
Optimization and tuning 56
Linking 58
Portability and migration 59
Compiler customization 60
Individual option descriptions 61
-### (-#) (pound sign) 62
-+ (plus sign) (C++ only) 63
--help (-qhelp) 63
--version (-qversion) 64
@*file* (-qoptfile) 66
-B 68
-C, -C! 69
-D 70
-E 71
-F 72
-I 74
-L 75
-O, -qoptimize 76
-P 79
-R 80
-S 81
-U 82
-X (-W) 83
-Werror (-qhalt) 84
-Wunsupported-xl-macro 85
-c 86
-dM (-qshowmacros) 87
-e 88
-fasm (-qasm) 88
-fcommon (-qcommon) 90
-fdollars-in-identifiers (-qdollar) 91
-fdump-class-hierarchy (-qdump_class_hierarchy) (C++ only) 92
-finline-functions (-qinline) 93
-fPIC (-qpic) 96
-fpack-struct (-qalign) 97
-fsigned-char, -funsigned-char (-qchars) 98
-fstandalone-debug 99
-fstrict-aliasing (-qalias=ansi), -qalias 99
-fsyntax-only (-qsyntaxonly) 101
-ftemplate-depth (-qtemplatedepth) (C++ only) 102
-ftrapping-math (-qfltrap) 103
-ftls-model (-qtls) 106
-ftime-report (-qphsinfo) 107

© Copyright IBM Corp. 1996, 2015

-funroll-loops (-qunroll), -funroll-all-loops (-qunroll=yes) . . . 108
-fvisibility (-qvisibility) . . . 110
-g . . . 112
-include (-qinclude) . . . 114
-isystem (-qc_stdinc) (C only) . . . 115
-isystem (-qcpp_stdinc) (C++ only) . . . 117
-isystem (-qgcc_c_stdinc) (C only) . . . 118
-isystem (-qgcc_cpp_stdinc) (C++ only) . . . 119
-l . . . 121
-maltivec (-qaltivec) . . . 122
-mcpu (-qarch) . . . 123
-mtune (-qtune) . . . 125
-o . . . 127
-p, -pg, -qprofile . . . 128
-qaggrcopy . . . 129
-qasm_as . . . 129
-qcache . . . 130
-qcheck . . . 133
-qcompact . . . 135
-qcrt, -nostartfiles (-qnocrt) . . . 136
-qdataimported, -qdatalocal, -qtocdata . . . 137
-qdirectstorage . . . 138
-qeh (C++ only) . . . 139
-qfloat . . . 139
-qfullpath . . . 143
-qfuncsect . . . 144
-qhot . . . 145
-qidirfirst . . . 147
-qignerrno . . . 148
-qinitauto . . . 149
-qinlglue . . . 151
-qipa . . . 152
-qisolated_call . . . 157
-qkeepparm . . . 159
-qlib, -nodefaultlibs (-qnolib) . . . 159
-qlibansi . . . 161
-qlinedebug . . . 161
-qlist . . . 162
-qlistfmt . . . 163
-qmaxmem . . . 166
-qmakedep, -MD (-qmakedep=gcc) . . . 167
-qpath . . . 169
-qpdf1, -qpdf2 . . . 170
-qprefetch . . . 177
-qpriority (C++ only) . . . 179
-qreport . . . 180
-qreserved_reg . . . 182
-qrestrict . . . 183
-qro . . . 184
-qroconst . . . 185
-qrtti, -fno-rtti (-qnortti) (C++ only) . . . 186
-qsaveopt . . . 187
-qshowpdf . . . 189
-qsimd . . . 190
-qsmallstack . . . 192
-qsmp . . . 193
-qspill . . . 196
-qstaticinline (C++ only) . . . 197
-qstdinc, -qnostdinc (-nostdinc, -nostdinc++) . . . 198
-qstrict . . . 199
-qstrict_induction . . . 204
-qtimestamps . . . 204
-qtmplinst (C++ only) . . . 205
-qxlcompatmacros . . . 206
-qunwind . . . 207
-r . . . 207
-s . . . 208
-shared (-qmkshrobj) . . . 209
-static (-qstaticlink) . . . 210
-std (-qlanglvl) . . . 212
-t . . . 216
-v, -V . . . 217
-w . . . 218
-x (-qsourcetype) . . . 219
-y . . . 221
Supported GCC options . . . 222

Chapter 5. Compiler pragmas reference . . . 227
Pragma directive syntax . . . 227
Scope of pragma directives . . . 227
Supported GCC pragmas . . . 228
Supported IBM pragmas . . . 228
#pragma disjoint . . . 229
#pragma execution_frequency . . . 230
#pragma ibm independent_loop . . . 231
#pragma nosimd . . . 232
#pragma option_override . . . 233
#pragma pack . . . 234
#pragma reachable . . . 238
#pragma simd_level . . . 238
#pragma STDC CX_LIMITED_RANGE . . . 239
#pragma unroll, #pragma nounroll . . . 240
Pragma directives for parallel processing . . . 242

Chapter 6. Compiler predefined macros . . . 263
General macros . . . 263
Macros indicating the XL C/C++ compiler . . . 264
Macros related to the platform . . . 266
Macros related to compiler features . . . 267
Macros related to compiler option settings . . . 267
Macros related to architecture settings . . . 269
Macros related to language levels . . . 269
Unsupported macros from other XL compilers . . . 270

Chapter 7. Compiler built-in functions 273
Fixed-point built-in functions . . . 273
Absolute value functions . . . 273
Assert functions . . . 274
Bit permutation functions . . . 274
Comparison functions . . . 274
Count zero functions . . . 275
Division functions . . . 275
Load functions . . . 276
Multiply functions . . . 277
Population count functions . . . 277
Rotate functions . . . 278
Store functions . . . 279
Trap functions . . . 280

Binary floating-point built-in functions 281
Absolute value functions 281
Conversion functions 281
FPSCR functions 284
Multiply-add/subtract functions 286
Reciprocal estimate functions 287
Rounding functions 287
Select functions 289
Square root functions 289
Software division functions 289
Store functions 290
Binary-coded decimal built-in functions 290
BCD add and subtract 291
BCD test add and subtract for overflow . . . 292
BCD comparison 292
BCD load and store 293
Synchronization and atomic built-in functions . . 294
Check lock functions 294
Clear lock functions 295
Compare and swap functions 296
Fetch functions 297
Load functions 298
Store functions 299
Synchronization functions 300
Cache-related built-in functions 301
Data cache functions 301
Prefetch built-in functions 303
Cryptography built-in functions 303
Advanced Encryption Standard functions . . . 303
Secure Hash Algorithm functions 306
Miscellaneous functions 307
Block-related built-in functions 309
__bcopy 309
Vector built-in functions 309
vec_abs 310
vec_abss 310
vec_add 311
vec_addc 312
vec_adds 312
vec_add_u128 313
vec_addc_u128 313
vec_adde_u128 314
vec_addec_u128 314
vec_all_eq 314
vec_all_ge 315
vec_all_gt 317
vec_all_in 318
vec_all_le 318
vec_all_lt 319
vec_all_nan 320
vec_all_ne 321
vec_all_nge 322
vec_all_ngt 323
vec_all_nle 323
vec_all_nlt 324
vec_all_numeric 324
vec_and 325
vec_andc 326
vec_any_eq 327
vec_any_ge 328
vec_any_gt 330
vec_any_le 331
vec_any_lt 332
vec_any_nan 333
vec_any_ne 334
vec_any_nge 335
vec_any_ngt 336
vec_any_nle 336
vec_any_nlt 337
vec_any_numeric 337
vec_any_out 338
vec_avg 338
vec_bperm 339
vec_ceil 339
vec_cipher_be 340
vec_cipherlast_be 340
vec_cmpb 340
vec_cmpeq 341
vec_cmpge 342
vec_cmpgt 343
vec_cmple 344
vec_cmplt 345
vec_cntlz 345
vec_cpsgn 346
vec_ctd 346
vec_ctf 347
vec_cts 347
vec_ctsl 348
vec_ctu 348
vec_ctul 349
vec_cvf 349
vec_div 350
vec_dss 350
vec_dssall 351
vec_dst 351
vec_dstst 351
vec_dststt 352
vec_dstt 352
vec_eqv 353
vec_expte 354
vec_extract 355
vec_floor 355
vec_gbb 356
vec_insert 356
vec_ld 357
vec_lde 358
vec_ldl 359
vec_loge 360
vec_lvsl 361
vec_lvsr 361
vec_madd 362
vec_madds 363
vec_max 363
vec_mergee 364
vec_mergeh 365
vec_mergel 365
vec_mergeo 366
vec_mfvscr 367
vec_min 367
vec_mladd 368
vec_mradds 369
vec_msub 369

vec_msum 370
vec_msums 371
vec_mtvscr 371
vec_mul 372
vec_mule 372
vec_mulo 373
vec_nabs 374
vec_nand 374
vec_ncipher_be 375
vec_ncipherlast_be 376
vec_nearbyint 376
vec_neg 377
vec_nmadd 377
vec_nmsub 378
vec_nor 378
vec_or 379
vec_orc 381
vec_pack 382
vec_packpx 383
vec_packs 383
vec_packsu 384
vec_perm 384
vec_pmsum_be 385
vec_popcnt 386
vec_promote 386
vec_re 387
vec_recipdiv 388
vec_revb 388
vec_reve 389
vec_rint 390
vec_rl 390
vec_round 391
vec_roundc 391
vec_roundm 392
vec_roundp 392
vec_roundz 393
vec_rsqrt 393
vec_rsqrte 394
vec_sbox_be 394
vec_sel 395
vec_shasigma_be 397
vec_sl 397
vec_sld 398
vec_sldw 399
vec_sll 400
vec_slo 400
vec_splat 401
vec_splats 402
vec_splat_s8 402
vec_splat_s16 403
vec_splat_s32 403
vec_splat_u8 404
vec_splat_u16 404
vec_splat_u32 405
vec_sqrt 405
vec_sr 406
vec_sra 406
vec_srl 407
vec_sro 408
vec_st 408
vec_ste 409
vec_stl 410
vec_sub 411
vec_sub_u128 412
vec_subc 412
vec_subc_u128 413
vec_sube_u128 413
vec_subec_u128 414
vec_subs 414
vec_sum2s 415
vec_sum4s 415
vec_sums 416
vec_trunc 416
vec_unpackh 416
vec_unpackl 417
vec_vclz 417
vec_vgbbd 418
vec_xl 419
vec_xl_be 421
vec_xld2 423
vec_xlds 424
vec_xlw4 424
vec_xor 425
vec_xst 426
vec_xst_be 427
vec_xstd2 428
vec_xstw4 429
GCC atomic memory access built-in functions (IBM extension) 430
Atomic lock, release, and synchronize functions 431
Atomic fetch and operation functions 432
Atomic operation and fetch functions 435
Atomic compare and swap functions 438
GCC object size checking built-in functions . . . 439
__builtin_object_size 439
__builtin___*_chk 440
Miscellaneous built-in functions 442
Optimization-related functions 442
Move to/from register functions 443
Memory-related functions 445
Transactional memory built-in functions 447
Transaction begin and end functions. 448
Transaction abort functions 449
Transaction inquiry functions 450
Transaction resume and suspend functions . . 454

Chapter 8. OpenMP runtime functions for parallel processing 455
omp_get_max_active_levels 455
omp_set_max_active_levels 455
omp_get_proc_bind 456
omp_get_schedule 456
omp_set_schedule 457
omp_get_thread_limit 457
omp_get_level 457
omp_get_ancestor_thread_num 458
omp_get_team_size 458
omp_get_active_level 458
omp_get_max_threads 458
omp_get_num_places 459
omp_get_num_procs 459
omp_get_num_threads 459

omp_set_num_threads 459
omp_get_partition_num_places 460
omp_get_partition_place_nums 460
omp_get_place_num 460
omp_get_place_num_procs 461
omp_get_place_proc_ids. 461
omp_get_thread_num 461
omp_in_final 462
omp_in_parallel 462
omp_set_dynamic 462
omp_get_dynamic. 462
omp_set_nested 463
omp_get_nested 463
omp_init_lock, omp_init_nest_lock 463
omp_destroy_lock, omp_destroy_nest_lock . . . 464
omp_set_lock, omp_set_nest_lock. 464
omp_unset_lock, omp_unset_nest_lock 464
omp_test_lock, omp_test_nest_lock 465
omp_get_wtime 465
omp_get_wtick 465

Notices 467
Trademarks 469

Index 471

About this document

This document is a reference for the IBM® XL C/C++ for Linux, V13.1.3 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: see the *XL C/C++ Installation Guide*.
- 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.

© Copyright IBM Corp. 1996, 2015

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 45 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 47 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 227 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 predefined macros,” on page 263 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, V13.1.3.

Chapter 7, “Compiler built-in functions,” on page 273 contains individual descriptions of XL C/C++ built-in functions for Power® architectures, categorized by their functionality.

Chapter 8, “OpenMP runtime functions for parallel processing,” on page 455 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, V13.1.3 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.
underlining	The default setting of a parameter of a compiler option or directive.	nomaf | maf

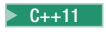
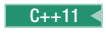
Table 1. Typographical conventions (continued)

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

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 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++ only ends	The text describes a feature that is supported in the C++ language only; or describes behavior that is specific to the C++ language.
IBM extension begins IBM extension ends	The text describes a feature that is an IBM extension to the standard language specifications.
C11 begins C11 ends	The text describes a feature that is introduced into standard C as part of C11.
C++11 begins   C++11 ends	The text describes a feature that is introduced into standard C++ as part of C++11.
C++14 begins C++14 ends	The text describes a feature that is introduced into standard C++ as part of C++14.

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):

- Optional items are shown below the main path:

- If you can choose from two or more items, they are shown vertically, in a stack.

  If you *must* choose one of the items, one item of the stack is shown on the main path.

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

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

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

- Keywords are shown in nonitalic letters and should be entered exactly as shown.

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

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 basic, or default, installation; these need little or no modification.

Related information

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

IBM XL C/C++ information

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

- Quick Start Guide

  The Quick Start Guide (`quickstart.pdf`) is intended to get you started with IBM XL C/C++ for Linux, V13.1.3. 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, V13.1.3 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_13.1.3/com.ibm.compilers.linux.doc/welcome.html.
- PDF documents

  PDF documents are available on the web at http://www.ibm.com/support/docview.wss?uid=swg27036675.

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

Table 3. XL C/C++ PDF files

Document title	PDF file name	Description
IBM XL C/C++ for Linux, V13.1.3 Installation Guide, GC27-6540-02	`install.pdf`	Contains information for installing XL C/C++ and configuring your environment for basic compilation and program execution.
Getting Started with IBM XL C/C++ for Linux, V13.1.3, GI13-2875-02	`getstart.pdf`	Contains an introduction to the XL C/C++ product, with information about setting up and configuring your environment, compiling and linking programs, and troubleshooting compilation errors.
IBM XL C/C++ for Linux, V13.1.3 Compiler Reference, SC27-6570-02	`compiler.pdf`	Contains information about the various compiler options, pragmas, macros, environment variables, and built-in functions.
IBM XL C/C++ for Linux, V13.1.3 Language Reference, SC27-6550-02	`langref.pdf`	Contains information about language extensions for portability and conformance to nonproprietary standards.
IBM XL C/C++ for Linux, V13.1.3 Optimization and Programming Guide, SC27-6560-02	`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 C/C++, see the C/C++ café at https://www.ibm.com/developerworks/community/groups/service/html/communityview?communityUuid=5894415f-be62-4bc0-81c5-3956e82276f3.

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 http://www.ibm.com/support/entry/portal/product/rational/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 http://www.ibm.com/software/products/en/xlcpp-linux.

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 9
- "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 allow for threadsafe compilations. You can use them to link the programs that use multithreading.

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/13.1.3/etc/xlc.cfg.*$OSRelease.gcc$gccVersion* file for your system. For example, /opt/ibm/xlC/13.1.3/etc/xlc.cfg.sles.12.gcc.4.8.2, /opt/ibm/xlC/13.1.3/etc/xlc.cfg.rhel.7.2.gcc.4.8.3, or /opt/ibm/xlC/13.1.3/etc/xlc.cfg.ubuntu.14.04.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

© Copyright IBM Corp. 1996, 2015

Table 4. Compiler invocations (continued)

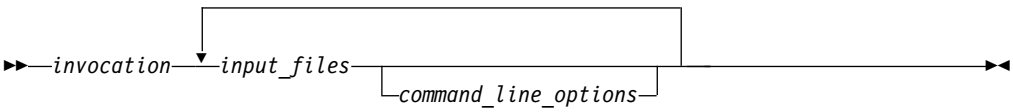
Invocations	Description	Equivalent invocations
c89	Invokes the compiler for C source files. This command supports all ANSI C89 language features, but does not support IBM language extensions. Use this invocation for strict conformance to the C89 standard.	c89_r
cc	Invokes the compiler for C source files. 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 212

Command-line syntax

You invoke the compiler using the following syntax, where *invocation* can be replaced with any valid XL C/C++ invocation command listed in Table 4 on page 1:



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`.

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 48

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 47

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 6.

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 6.

Specifying compiler options in a configuration file

The default configuration file (/opt/ibm/xlC/13.1.3/etc/xlc.cfg.*$OSRelease.gcc$gccVersion*, for example, /opt/ibm/xlC/13.1.3/etc/xlc.cfg.sles.12.gcc.4.8.3, /opt/ibm/xlC/13.1.3/etc/xlc.cfg.rhel.7.2.gcc.4.8.5, or /opt/ibm/xlC/13.1.3/etc/xlc.cfg.ubuntu.14.04.gcc.4.8.2) defines values and compiler options for the compiler. The compiler refers to this file when compiling C or C++ programs.

The configuration file is a plain text file. You can edit this file, or create an additional customized configuration file to support specific compilation requirements. For more information, see "Using custom compiler configuration files" on page 37.

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 61, 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 227.

Other pragmas do not have equivalent command-line options; these are described in detail throughout Chapter 5, "Compiler pragmas reference," on page 227.

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 71	Preprocesses the source files and writes the output to standard output. By default, `#line` directives are generated.
"-P" on page 79	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 69	Preserves comments in preprocessed output.
"-D" on page 70	Defines a macro name from the command line, as if in a `#define` directive.

Option	Description
-dD[1]	Emits macro definitions to preprocessed output and prints the output.
"-dM (-qshowmacros)" on page 87[1]	Emits macro definitions to preprocessed output.
"-qmakedep, -MD (-qmakedep=gcc)" on page 167	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 82	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 74
- "-isystem (-qc_stdinc) (C only)" on page 115
- "-isystem (-qcpp_stdinc) (C++ only)" on page 117
- "-isystem (-qgcc_c_stdinc) (C only)" on page 118
- "-isystem (-qgcc_cpp_stdinc) (C++ only)" on page 119
- "-qidirfirst" on page 147
- "-include (-qinclude)" on page 114
- "-qstdinc, -qnostdinc (-nostdinc, -nostdinc++)" on page 198

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 58.

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 -Blinker_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 209

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 58
- "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 http://www.ibm.com/support/entry/portal/product/rational/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/13.1.3/lib/libxl.a` `/opt/ibm/xlC/13.1.3/lib/libxlopt.a`	XL C/C++ compiler libraries
vacpp.rte	`/opt/ibmcmp/vac/13.1.3/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"
- "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 84
- "Listings, messages, and compiler information" on page 55

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 **-qpdf1** phase. 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 **-qpdf1** phase. The warning message is also issued when the optimization level changes from the **-qpdf1** phase to the **-qpdf2** phase.

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 55

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.

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 37.

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

© Copyright IBM Corp. 1996, 2015

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* and *NLSPATH*, which must be set if you are using a locale other than the default en_US, all of these variables are optional.

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

LD_RUN_PATH
Specifies search paths for dynamically loaded libraries, equivalent to using the **-R** link-time option. The shared-library locations named by the environment variable are embedded into the executable, so the dynamic linker can locate the libraries at application run time. For more information about this environment variable, see your operating system documentation. See also "-R" on page 80.

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

PATH Specifies the directory search path for the executable files of the compiler. Executables are in /opt/ibm/xlC/13.1.3/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 37.

Runtime environment variables

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

LD_LIBRARY_PATH
Specifies an alternate directory search path for dynamically linked libraries at application run time. If shared libraries required by your application have been moved to an alternate directory that was not specified at link

time, and you do not want to relink the executable, you can set this environment variable to allow the dynamic linker to locate them at run time. For more information about this environment variable, see your operating system documentation.

PDFDIR
Optionally specifies the directory in which profiling information is saved when you run an application that you have compiled with the **-qpdf1** option. The default value is unset, and the compiler places the profile data file in the current working directory. 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 170 for more information.

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_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_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.

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.

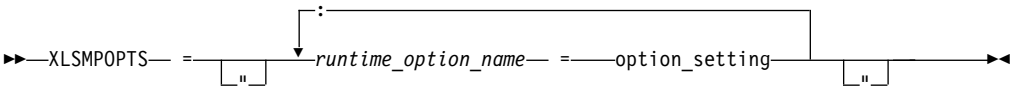
If you are using OpenMP constructs for parallelization, you can also specify runtime options using the OMP environment variables, as discussed in "Environment variables for OpenMP" on page 24.

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

Related information
- "Pragma directives for parallel processing" on page 242

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:



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 are as follows:

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 are as follows:

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

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

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

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 193.

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.

Performance tuning options are as follows:

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

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

particular application highly responsive, but can also harm the overall responsiveness of the system unless the thread is given instructions to periodically scan for and yield to requests from other applications.

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

The default value for *num* is 100.

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

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

The default value for *num* is 100.

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

The default value for *num* is 500.

Dynamic profiling options are as follows:

profilefreq=*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 35

-qsmp

Related information:

"OMP_PLACES" on page 29

Environment variables for OpenMP

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

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

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

OMP_DISPLAY_ENV: When a program that uses the OpenMP runtime is invoked and the **OMP_DISPLAY_ENV** environment variable is set, the OpenMP runtime 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:

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 29

"OMP_PROC_BIND" on page 31

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

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 29

"OMP_PROC_BIND" on page 31

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.

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:

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** routine 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:

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.

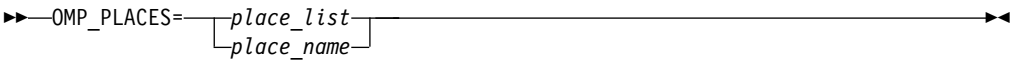
Example

```
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:

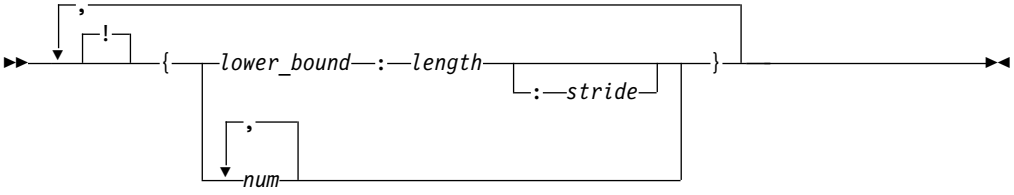
- An explicit list of places that are described by non-negative numbers
- An abstract name that describes a set of places

OMP_PLACES syntax

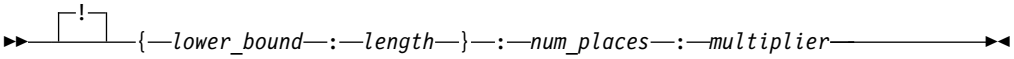


where *place_list* takes one of the following syntax forms:

place_list syntax: form 1

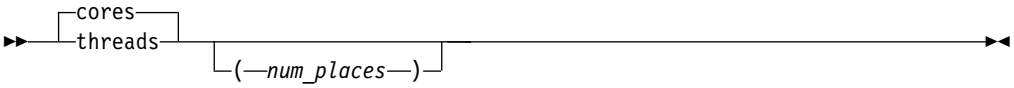


place_list syntax: form 2



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



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

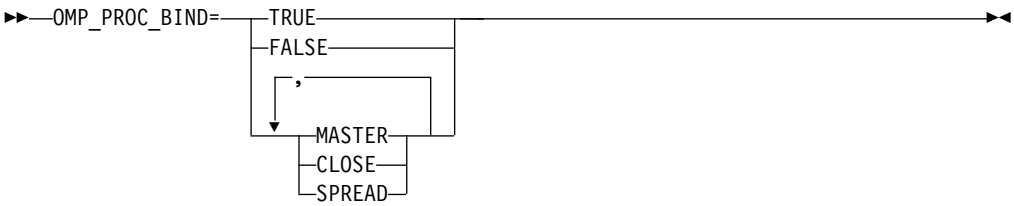
When requested places are fewer than that are available on the system, the execution environment assigns places in the order of the place list at run time. When requested places are more than that are available on the system, the execution environment assigns the maximum number of places that the system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



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 456

Related information:

"XLSMPOPTS" on page 20

"OMP_PLACES" on page 29

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 457

"omp_get_schedule" on page 456

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:

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.

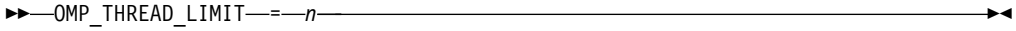
```
export OMP_STACKSIZE="10M"
export OMP_STACKSIZE=" 10 M "
```

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.



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_PLACES" on page 29

"OMP_NUM_THREADS" on page 28

"OMP_NESTED" on page 27

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:

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/13.1.3/etc/xlc.cfg.*$OSRelease.gcc$gccVersion* at installation time (for example, /opt/ibm/xlC/13.1.3/etc/xlc.cfg.sles.12.gcc.4.8.2, /opt/ibm/xlC/13.1.3/etc/xlc.cfg.rhel.7.2.gcc.4.8.3, or /opt/ibm/xlC/13.1.3/etc/xlc.cfg.ubuntu.14.04.gcc.4.8.2). (See the *XL C/C++ Installation Guide* for more

information on the various tools you can use to generate the configuration file during installation.) The configuration file specifies information that the compiler uses when you invoke it.

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

If you want users to be able to choose among several sets of compiler options, you might want to use custom configuration files for specific needs. For example, you might want to enable **-qlist** by default for compilations using the **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 72

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>
```

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

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

DEFLT:
    options=<C>
```

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

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

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

Default configuration file xlc.cfg

```
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, V13.1.3 with the Advance Toolchain

IBM XL C/C++ for Linux, V13.1.3 supports IBM Advance Toolchain 9.0, which is a set of open source development tools and runtime libraries. With IBM Advance Toolchain 9.0, you can take advantage of the latest POWER® hardware features on Linux, especially the tuned libraries. For more information about the Advance Toolchain 9.0, see IBM Advance Toolchain for PowerLinux™ Documentation.

To use IBM XL C/C++ for Linux, V13.1.3 with the Advance Toolchain, take the following steps:

1. Install the **at9.0** packages into the default installation location. For instructions, see IBM Advance Toolchain for PowerLinux Documentation.
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.

   - If you installed the compiler in the default location, issue the following command:

     ```
     xlc_configure -at
     ```

   - If you installed the compiler in a nondefault installation (NDI) location, issue the following command:

     ```
     xlc_configure -at -ibmcmp $ndi_path
     ```

     where *$ndi_path* is the directory in which you installed the compiler.
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/13.1.3/bin/xlc_at
     /opt/ibm/xlC/13.1.3/bin/xlC_at
     ```

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

     ```
     $ndi_path/xlC/13.1.3/bin/xlc_at
     $ndi_path/xlC/13.1.3/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/13.1.3/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/13.1.3/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.

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 72 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 45

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.

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.

In IBM XL C/C++ for Linux, V13.1.3, SLM Tags logging is provided for evaluation purposes only, and logging is enabled only when you specify the **-qxflag=slmtags** compiler option to invoke the license metric logging. When logging is enabled, the compiler logs compiler license usage in the SLM Tags format, to files in the */user_home*/xl-slmtags directory, where */user_home* is the user's home directory. 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.

Setting up SLM Tags logging

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

Procedure

1. Determine which user IDs are from authorized users.
2. Create a file with the name XLAuthorizedUsers in the /etc directory. The file contains information for authorized users, one line for each user. Each line should contain only 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.

   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:

© Copyright IBM Corp. 1996, 2015

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

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

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

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

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

What to do next

SLM Tags logging is enabled when you specify the **-qxflag=slmtags** option. You can add this option to the compiler invocation command for a given invocation. If you want all compiler invocations to have SLM Tags logging enabled, you can add this option to the appropriate stanza in your compiler configuration file.

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 each time the compiler is invoked with the **-qxflag=slmtags** option. 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.

Related information:

IBM License Metric Tool (ILMT)

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 48
- "Language element control" on page 49
- "Template control (C++ only)" on page 50
- "Floating-point and integer control" on page 50
- "Error checking and debugging" on page 52
- "Listings, messages, and compiler information" on page 55
- "Optimization and tuning" on page 56
- "Object code control" on page 51
- "Linking" on page 58
- "Portability and migration" on page 59
- "Compiler customization" on page 60

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 86	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 69	When used in conjunction with the **-E** or **-P** options, preserves or removes comments in preprocessed output.
"-dM (-qshowmacros)" on page 87	Emits macro definitions to preprocessed output.
"-E" on page 71	Preprocesses the source files named in the compiler invocation, without compiling.

© Copyright IBM Corp. 1996, 2015

Table 9. Compiler output options (continued)

Option name	Description
"-o" on page 127	Specifies a name for the output object, assembler, executable, or preprocessed file.
"-P" on page 79	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 167	Produces the dependency files that are used by the **make** tool for each source file.
"-qtimestamps" on page 204	Controls whether or not implicit time stamps are inserted into an object file.
"-shared (-qmkshrobj)" on page 209	Creates a shared object from generated object files.
"-S" on page 81	Generates an assembler language file for each source file.
"-X (-W)" on page 83	**-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 114	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 74	Adds a directory to the search path for include files.

Table 10. Compiler input options (continued)

Option name	Description
"-qidirfirst" on page 147	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 198	Specifies whether the standard include directories are included in the search paths for system and user header files.
"-x (-qsourcetype)" on page 219	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 70	Defines a macro as in a `#define` preprocessor directive.
"-fasm (-qasm)" on page 88	Controls the interpretation and subsequent generation of code for assembler language extensions.
"-maltivec (-qaltivec)" on page 122	Enables the compiler support for vector data types and operators.
"-fdollars-in-identifiers (-qdollar)" on page 91	Allows the dollar-sign ($) symbol to be used in the names of identifiers.
"-qstaticinline (C++ only)" on page 197	Controls whether inline functions are treated as having `static` or `extern` linkage.
"-std (-qlanglvl)" on page 212	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 82	Undefines a macro defined by the compiler or by the **-D** compiler option.
"-X (-W)" on page 83	**-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 102	Specifies the maximum number of recursively instantiated template specializations that will be processed by the compiler.
"-qtmplinst (C++ only)" on page 205	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 98	Determines whether all variables of type `char` is treated as `signed` or `unsigned`.
"-qfloat" on page 139	Selects different strategies for speeding up or improving the accuracy of floating-point calculations.
"-qstrict" on page 199	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.
"-y" on page 221	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 90	Controls where uninitialized global variables are allocated.
"-qeh (C++ only)" on page 139	Controls whether exception handling is enabled in the module being compiled.
"-qfuncsect" on page 144	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 151	When used with **-O2** or higher optimization, inlines glue code that optimizes external function calls in your application.
"-qpriority (C++ only)" on page 179	Specifies the priority level for the initialization of static objects.
"-qreserved_reg" on page 182	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 184	Specifies the storage type for string literals.
"-qroconst" on page 185	Specifies the storage location for constant values.
"-qrtti, -fno-rtti (-qnortti) (C++ only)" on page 186	Generates runtime type identification (RTTI) information for exception handling and for use by the `typeid` and `dynamic_cast` operators.
"-qsaveopt" on page 187	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.
"-r" on page 207	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 208	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
- -fPIE, -fno-PIE
- -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 55.

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 62	Previews the compilation steps specified on the command line, without actually invoking any compiler components.
"-fstandalone-debug" on page 99	When used with the **-g** option, controls whether to generate the debugging information for all symbols.
"-fsyntax-only (-qsyntaxonly)" on page 101	Performs syntax checking without generating an object file.
"-g" on page 112	Generates debugging information for use by a symbolic debugger, and makes the program state available to the debugging session at selected source locations.
"-qcheck" on page 133	Generates code that performs certain types of runtime checking.
"-ftrapping-math (-qflttrap)" on page 103	Determines what types of floating-point exceptions to detect at run time.
"-qfullpath" on page 143	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 149	Initializes uninitialized automatic variables to a specific value, for debugging purposes.
"-qkeepparm" on page 159	When used with **-O2** or higher optimization, specifies whether procedure parameters are stored on the stack.

Table 15. Error checking and debugging options (continued)

Option name	Description
"-qlinedebug" on page 161	Generates only line number and source file name information for a debugger.
"-Werror (-qhalt)" on page 84	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 85	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
- -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 52 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 92	Dumps a representation of the hierarchy and virtual function table layout of each class object to a file.

Table 16. Listings and messages options (continued)

Option name	Description
"-qlist" on page 162	Produces a compiler listing file that includes object and constant area sections.
"-qlistfmt" on page 163	Creates a report in XML or HTML format to help you find optimization opportunities.
"-qreport" on page 180	Produces listing files that show how sections of code have been optimized.
"--help (-qhelp)" on page 63	Displays the man page of the compiler.
"--version (-qversion)" on page 64	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 93	Attempts to inline functions instead of generating calls to those functions, for improved performance.
"-fstrict-aliasing (-qalias=ansi), -qalias" on page 99	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 108	Controls loop unrolling, for improved performance. **Equivalent pragma:** #pragma unroll
"-fvisibility (-qvisibility)" on page 110	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 123	Specifies the processor architecture for which the code (instructions) should be generated.

Table 17. Optimization and tuning options (continued)

Option name	Description
-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 76	Specifies whether to optimize code during compilation and, if so, at which level.
"-p, -pg, -qprofile" on page 128	Prepares the object files produced by the compiler for profiling.
"-qaggrcopy" on page 129	Enables destructive copy operations for structures and unions.
"-qcache" on page 130	Specifies the cache configuration for a specific execution machine.
"-qcompact" on page 135	Avoids optimizations that increase code size.
"-qdataimported, -qdatalocal, -qtocdata" on page 137	Marks data as local or imported.
"-qdirectstorage" on page 138	Informs the compiler that a given compilation unit may reference write-through-enabled or cache-inhibited storage.
"-qhot" on page 145	Performs high-order loop analysis and transformations (HOT) during optimization.
"-qignerrno" on page 148	Allows the compiler to perform optimizations as if system calls would not modify `errno`.
"-qipa" on page 152	Enables or customizes a class of optimizations known as interprocedural analysis (IPA).
"-qisolated_call" on page 157	Specifies functions in the source file that have no side effects other than those implied by their parameters.
"-qlibansi" on page 161	Assumes that all functions with the name of an ANSI C library function are in fact the system functions.
"-qmaxmem" on page 166	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 170	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 177	Inserts prefetch instructions automatically where there are opportunities to improve code performance.

Table 17. Optimization and tuning options (continued)

Option name	Description
"-qrestrict" on page 183	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 189	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 190	Controls whether the compiler can automatically take advantage of vector instructions for processors that support them. **Equivalent pragma:** #pragma nosimd
"-qsmallstack" on page 192	Minimizes stack usage where possible. Disables optimizations that increase the size of the stack frame.
"-qsmp" on page 193	Enables parallelization of program code.
"-qstrict" on page 199	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 204	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.
"-qunwind" on page 207	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 88	When used together with the **-shared (-qmkshrobj)** option , specifies an entry point for a shared object.

Table 18. Linking options (continued)

Option name	Description
"-L" on page 75	At link time, searches the directory path for library files specified by the **-l** option.
"-l" on page 121	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 136	Specifies whether system startup files are to be linked.
"-qlib, -nodefaultlibs (-qnolib)" on page 159	Specifies whether standard system libraries and XL C/C++ libraries are to be linked.
"-R" on page 80	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 210	Controls whether static or shared runtime libraries are linked into an application.
"-X (-W)" on page 83	**-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 97	Specifies the alignment of data objects in storage, which avoids performance problems with misaligned data.

Table 19. Portability and migration options (continued)

Option name	Description
"-qxlcompatmacros" on page 206	Defines the following legacy macros: C++ `__IBMCPP__`, `__xlC__`, `__xlC_ver__` C++, C `__IBMC__`, and `__xlc__` C. This option helps you migrate programs from IBM XL C/C++ for Linux for big endian distributions to IBM XL C/C++ for Linux V13.1.3 for little endian distributions.

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 66	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 68	Specifies substitute path names for XL C/C++ components such as the assembler, C preprocessor, and linker.
"-F" on page 72	Names an alternative configuration file or stanza for the compiler.
"-isystem (-qc_stdinc) (C only)" on page 115	Changes the standard search location for the XL C header files.
"-isystem (-qcpp_stdinc) (C++ only)" on page 117	Changes the standard search location for the XL C++ header files.
"-isystem (-qgcc_c_stdinc) (C only)" on page 118	Changes the standard search location for the GNU C system header files.
"-isystem (-qgcc_cpp_stdinc) (C++ only)" on page 119	Changes the standard search location for the GNU C++ system header files.
"-qasm_as" on page 129	Specifies the path and flags used to invoke the assembler in order to handle assembler code in an `asm` assembly statement.
"-qpath" on page 169	Specifies substitute path names for XL C/C++ components such as the compiler, assembler, linker, and preprocessor.
"-qspill" on page 196	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 216	Applies the prefix specified by the **-B** option to the designated components.

Table 20. Compiler customization options (continued)

Option name	Description
"-X (-W)" on page 83	Passes the listed options to a component that is executed during compilation.

Individual option descriptions

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

For each option, the following information is provided:

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

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

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 227

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

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

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

Predefined macros
Many compiler options set macros that are protected (that is, cannot be undefined or redefined by the user). Where applicable, any macros that are predefined by the option, and the values to which they are defined, are listed in this section. A reference list of these macros (as well as others that

are defined independently of option setting) is provided in Chapter 6, "Compiler predefined macros," on page 263

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

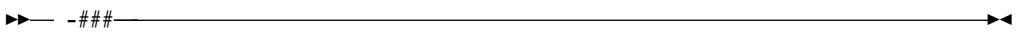
None.

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 217

-+ (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 219

--help (-qhelp)

Category

Listings, messages, and compiler information

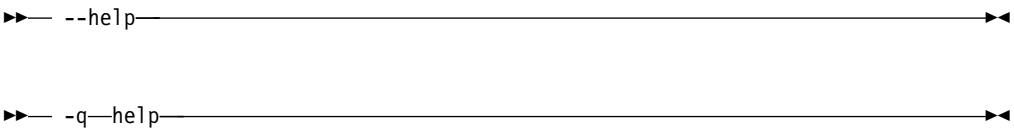
Pragma equivalent

None.

Purpose

Displays the man page of the compiler.

Syntax



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

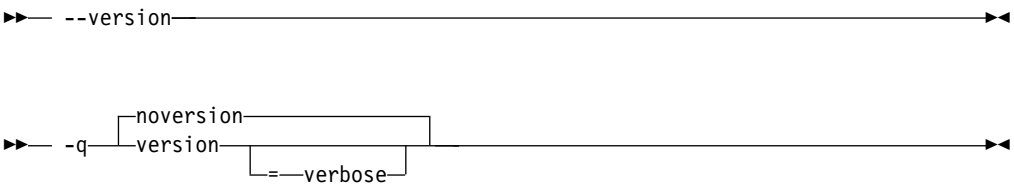
Pragma equivalent

None.

Purpose

Displays the version and release of the compiler being invoked.

Syntax



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_name*Version: *VV.RR.MMMM.LLLL*

where:

V Represents the version.
R Represents the release.
M Represents the modification.
L Represents the level.

For more details, see Example 1.

-qversion=verbose shows component information in the following format:

component_name Version: *VV.RR*(*product_name*) Level: *component_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, V13.1.3 (5765-J08; 5725-C73)
Version: 13.01.0002.0000
```

Example 2

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

```
IBM XL C/C++ for Linux, V13.1.3 (5765-J08; 5725-C73)
Version: 13.01.0003.0000
Driver Version: 13.1.3(C/C++) Level: 150508
ID: _hnbfIvWfEeSjz7qEhQiYJQ
C Front End Version: 15.1.3(Fortran) Level: 150506
ID: _EwaE2-iLEeSbzZ-i2Itj4A
C++ Front End Version: 13.1.3(C/C++) Level: 150511
ID: _YU-wovhCEeSjz7qEhQiYJQ
High-Level Optimizer Version: 13.1.3(C/C++) and 15.1.3(Fortran)
Level: 150512 ID: _mSHAgvkLEeSjz7qEhQiYJQ
Low-Level Optimizer Version: 13.1.3(C/C++) and 15.1.3(Fortran)
Level: 150511 ID: _YY5AQvhCEeSjz7qEhQiYJQ
```

Related information

- "-qsaveopt" on page 187

@*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

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 187

-B

Category

Compiler customization

Pragma equivalent

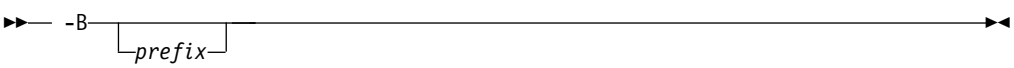
None.

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



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 216 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/13.1.3/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 169
- "-t" on page 216
- "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

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 71
- "-P" on page 79

-D

Category

Language element control

Pragma equivalent

None.

Purpose

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

Syntax

Defaults

Not applicable.

Parameters

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

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

Usage

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

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

Predefined macros

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

Examples

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

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

Related information

- "-U" on page 82
- Chapter 6, "Compiler predefined macros," on page 263

-E

Category

Output control

Pragma equivalent

None.

Purpose

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

Syntax

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 69
- "-P" on page 79
- "-fsyntax-only (-qsyntaxonly)" on page 101

-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

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 37
- "-B" on page 68
- "-t" on page 216
- "-X (-W)" on page 83
- "Specifying compiler options in a configuration file" on page 5
- "Compile-time and link-time environment variables" on page 18

-I

Category

Input control

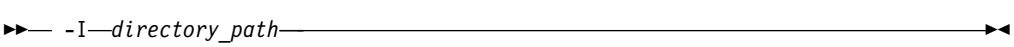
Pragma equivalent

None.

Purpose

Adds a directory to the search path for include files.

Syntax



Defaults

See "Directory search sequence for included files" on page 8 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 147
- "-qstdinc, -qnostdinc (-nostdinc, -nostdinc++)" on page 198
- "-include (-qinclude)" on page 114
- "Directory search sequence for included files" on page 8
- "Specifying compiler options in a configuration file" on page 5

-L

Category

Linking

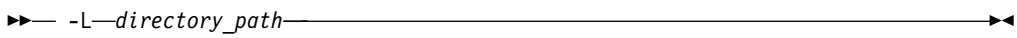
Pragma equivalent

None.

Purpose

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

Syntax



Defaults

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

Parameters

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

Usage

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

If the **-L***directory* option is specified both in the configuration file and on the command line, search paths specified in the configuration file are the first to be searched at link time.

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 121
- "-R" on page 80

-O, -qoptimize

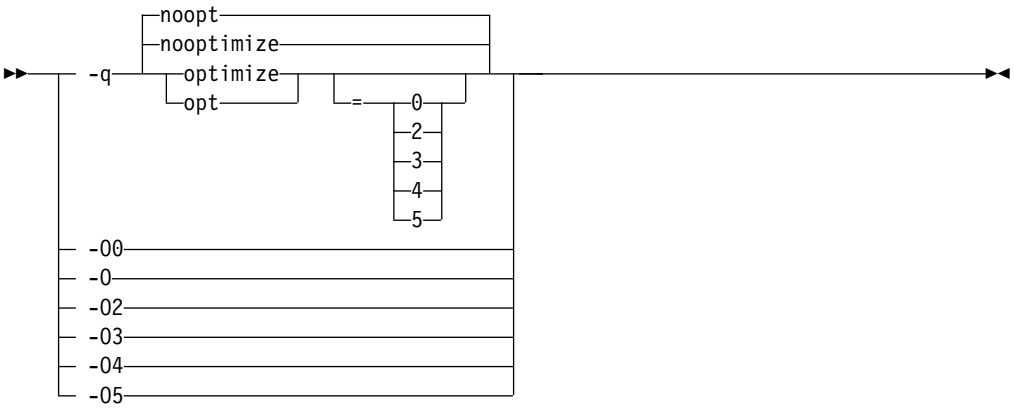
Category

Optimization and tuning

Purpose

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

Syntax



Defaults

-qnooptimize or **-O0** or **-qoptimize=0**

Parameters

-O0 | nooptimize | noopt | optimize|opt=0
Performs only quick local optimizations such as constant folding and elimination of local common subexpressions.

This setting implies **-qstrict_induction** unless **-qnostrict_induction** is explicitly specified.

-O | -O2 | optimize | opt | optimize|opt=2
Performs optimizations that the compiler developers considered the best combination for compilation speed and runtime performance. The optimizations may change from product release to release. If you need a specific level of optimization, specify the appropriate numeric value.

This setting implies **-qstrict** and **-qnostrict_induction**, unless explicitly negated by **-qstrict_induction** or **-qnostrict**.

-O3 | optimize|opt=3
Performs additional optimizations that are memory intensive, compile-time intensive, or both. They are recommended when the desire for runtime improvement outweighs the concern for minimizing compilation resources.

-O3 applies the **-O2** level of optimization, but with unbounded time and memory limits. **-O3** also performs higher and more aggressive optimizations that have the potential to slightly alter the semantics of your program. The compiler guards against these optimizations at **-O2**. The aggressive optimizations performed when you specify **-O3** are:

1. 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.
2. Specifying **-O3** implies **-qhot=level=0**, unless you explicitly specify **-qhot** or **-qhot=level=1** option.

-qfloat=rsqrt is set by default with **-O3**.

-qmaxmem=-1 is set by default with **-O3**, allowing the compiler to use as much memory as necessary when performing optimizations.

Built-in functions do not change `errno` at **-O3**.

Integer divide instructions are considered too dangerous to optimize even at **-O3**.

Refer to "-ftrapping-math (-qfltrap)" on page 103 to see the behavior of the compiler when you specify **optimize** options with the **-ftrapping-math (-qfltrap)** 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.

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 145
- “-qipa” on page 152
- “-qpdf1, -qpdf2” on page 170
- “-qstrict” on page 199
- "Optimizing your applications" in the *XL C/C++ Optimization and Programming Guide*.
- “#pragma option_override” on page 233

-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

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 69
- "-E" on page 71
- "-fsyntax-only (-qsyntaxonly)" on page 101

-R

Category

Linking

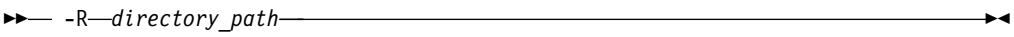
Pragma equivalent

None.

Purpose

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

Syntax



Defaults

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

Usage

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

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

Predefined macros

None.

Examples

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

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

Related information

- "-L" on page 75

-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

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 71
- "-P" on page 79

-U

Category

Language element control

Pragma equivalent

None.

Purpose

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

Syntax

Defaults

Many macros are predefined by the compiler; see Chapter 6, "Compiler predefined macros," on page 263 for those that can be undefined (that is, are not *protected*). The compiler configuration file also uses the **-D** option to predefine several macro names for specific invocation commands; for details, see the configuration file for your system.

Parameters

name
The macro you want to undefine.

Usage

The **-U** option is *not* equivalent to the `#undef` preprocessor directive. It *cannot* undefine names defined in the source by the `#define` preprocessor directive. It can only undefine names defined by the compiler or by the **-D** option.

The **-U***name* option has a higher precedence than the **-D***name* option.

Predefined macros

None.

Examples

Assume that your operating system defines the name `__unix`, but you do not want your compilation to enter code segments conditional on that name being defined, compile `myprogram.c` so that the definition of the name `__unix` is nullified by entering:

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

Related information
- "-D" on page 70

-X (-W)

Category

Compiler customization

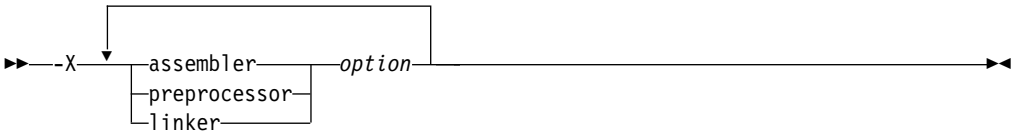
Pragma equivalent

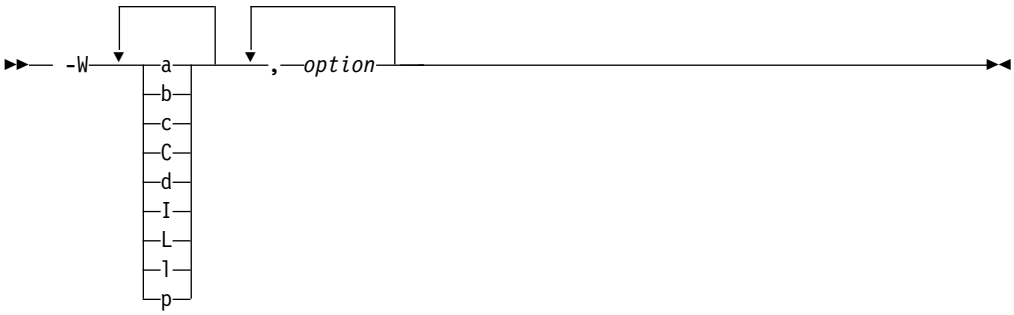
None.

Purpose

Passes the listed options to a component that is executed during compilation.

Syntax





Parameters

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

Note: 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
a	assembler	The assembler	as
b		The low-level optimizer	xlCcode

Parameter of -W	Parameter of -X	Description	Component name
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
p	preprocessor	The preprocessor	xlCentry

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**.

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 the following command:

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

Related information

- "Invoking the compiler" on page 1

-Werror (-qhalt)

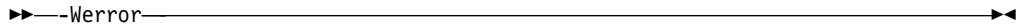
Category

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



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

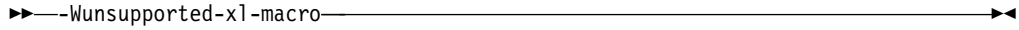
Pragma equivalent

None.

Purpose

Checks whether any unsupported XL macro is used.

Syntax



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, V13.1.3.

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 270

"-qxlcompatmacros" on page 206

-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

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 71
- "-o" on page 127
- "-P" on page 79
- "-fsyntax-only (-qsyntaxonly)" on page 101

-dM (-qshowmacros)

Category

"Output control" on page 47

Pragma equivalent

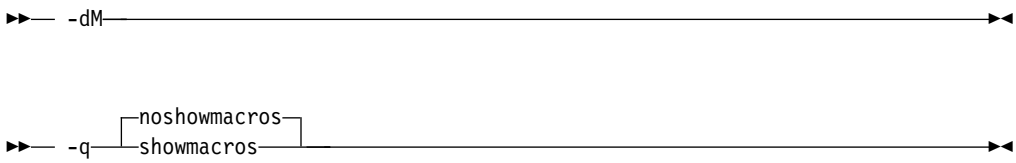
None

Purpose

Emits macro definitions to preprocessed output.

Emitting macros to preprocessed output can help determine functionality available in the compiler. The macro listing may prove useful for debugging complex macro expansions, as well.

Syntax



Defaults

-qnoshowmacros

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 71
- "-P" on page 79

-e

Category

Linking

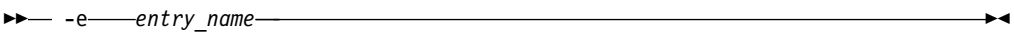
Pragma equivalent

None.

Purpose

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

Syntax



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 209

-fasm (-qasm)

Category

Language element control

Pragma equivalent

None.

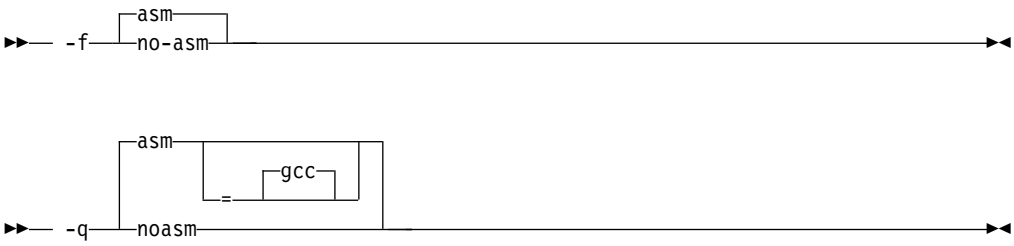
Purpose

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



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 129
- “-std (-qlanglvl)” on page 212
- "Inline assembly statements" in the *XL C/C++ Language Reference*

-fcommon (-qcommon)

Category

Object code control

Pragma equivalent

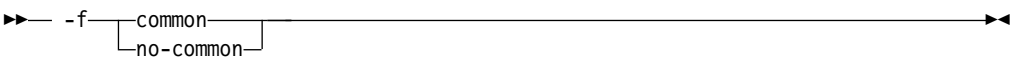
None.

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



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 209
- "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

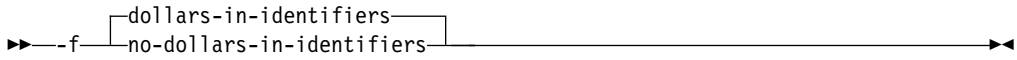
None

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



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 212

-fdump-class-hierarchy (-qdump_class_hierarchy) (C++ only)

Category

Listings, messages, and compiler information

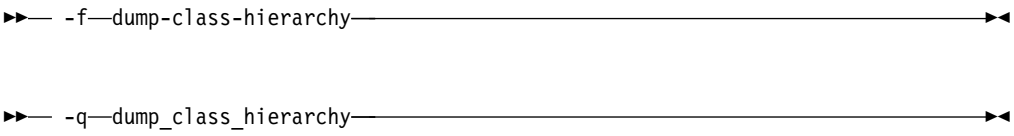
Pragma equivalent

None.

Purpose

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

Syntax



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
```

-finline-functions (-qinline)

Category

Optimization and tuning

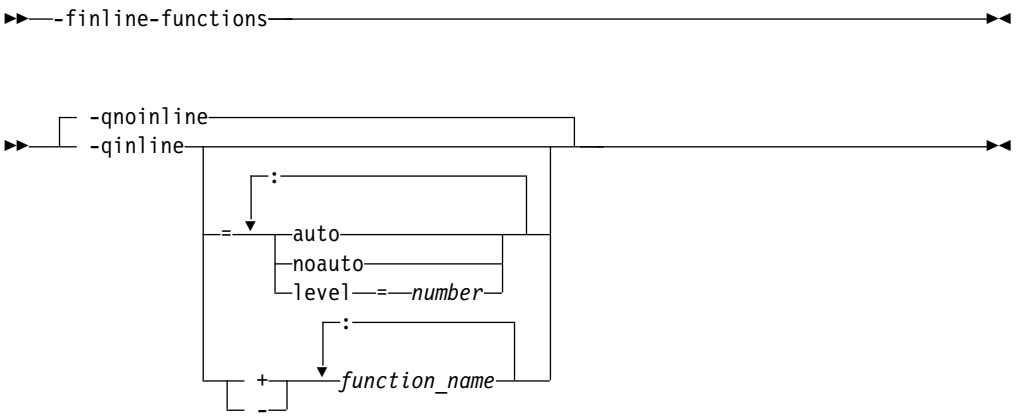
Pragma equivalent

None.

Purpose

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

Syntax



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

C

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

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 112
- "-qipa" on page 152
- "-O, -qoptimize" on page 76
- "Compiler listings" on page 13
- "always_inline (IBM extension)" in the *XL C/C++ Language Reference*

-fPIC (-qpic)

Category

Object code control

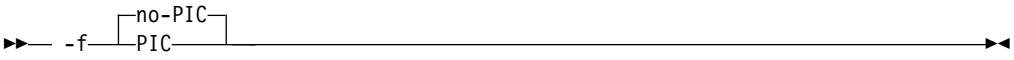
Pragma equivalent

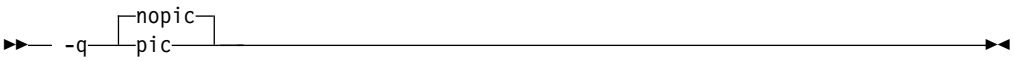
None.

Purpose

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

Syntax





Defaults

- , or **-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 209

-fpack-struct (-qalign)

Category

Portability and migration

Purpose

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

Syntax

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 228
- "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*

-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

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

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 112

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

Category

Optimization and tuning

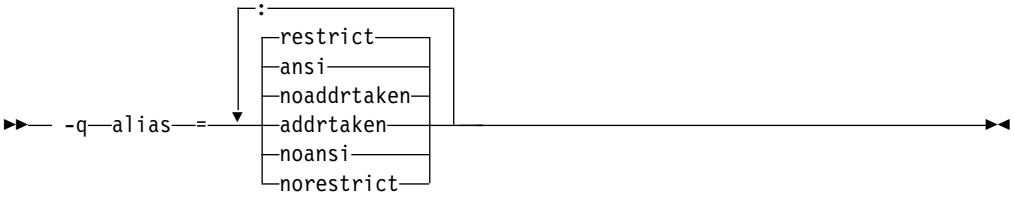
Pragma equivalent

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



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.

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

-qalias=restrict is independent from other **-qalias** suboptions. Using the **-qalias=restrict** option 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 152
- *The may_alias type attribute (IBM extension)* in the *XL C/C++ Language Reference*
- "-qrestrict" on page 183

-fsyntax-only (-qsyntaxonly)

Category

Error checking and debugging

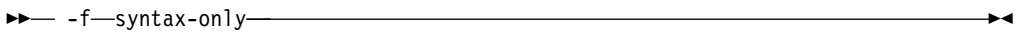
Pragma equivalent

None.

Purpose

Performs syntax checking without generating an object file.

Syntax



Defaults

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

Usage

The **-P**, **-E**, and **-C** options override the **-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 69
- "-c" on page 86
- "-E" on page 71
- "-o" on page 127
- "-P" on page 79

-ftemplate-depth (-qtemplatedepth) (C++ only)

Category

Template control

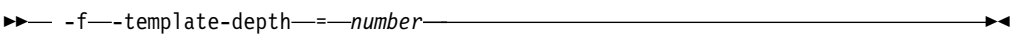
Pragma equivalent

None.

Purpose

Specifies the maximum number of recursively instantiated template specializations that will be processed by the compiler.

Syntax



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 (-qfltrap)

Category

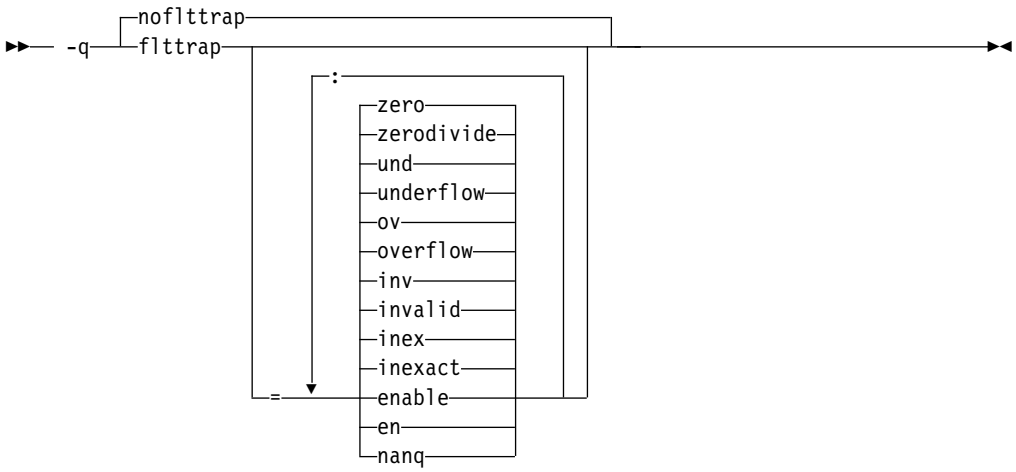
Error checking and debugging

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



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 139
- "-mcpu (-qarch)" on page 123

-ftls-model (-qtls)

Category

Object code control

Pragma equivalent

None.

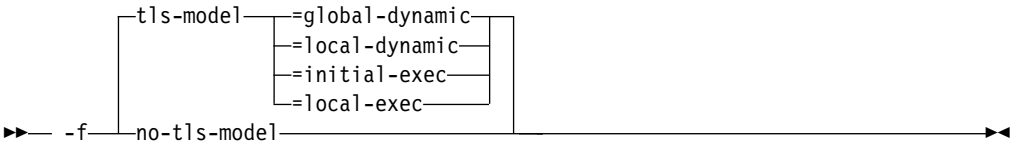
Purpose

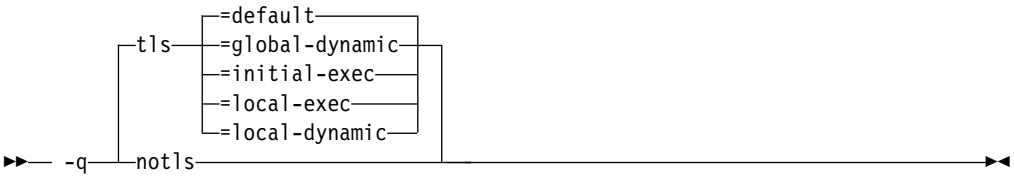
Enables recognition of the `__thread` storage class specifier, which designates variables that are to be allocated thread-local storage; and specifies the threadlocal storage model to be used.

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





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 96
- "The __thread storage class specifier" in the *XL C/C++ Language Reference*

-ftime-report (-qphsinfo)

Category

Listings, messages, and compiler information

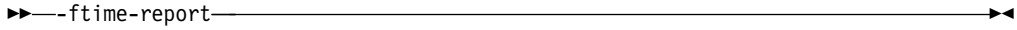
Pragma equivalent

None.

Purpose

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

Syntax



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 `-qphsinfo` 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.

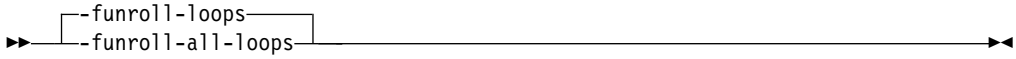
-funroll-all-loops
Instructs the compiler to search for more opportunities for loop unrolling than that performed with **-funroll-loops**. In general, **-funroll-all-loops** has more

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



Option syntax

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 240

-fvisibility (-qvisibility)

Category

Optimization and tuning

Pragma equivalent

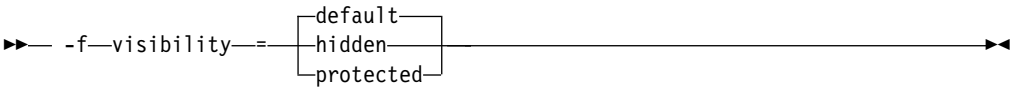
- **-fvisibility**: #pragma GCC visibility push (default | protected | hidden)
- **-qvisibility**: #pragma GCC visibility push (default | protected | hidden)

#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



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 209
- "Supported GCC pragmas" on page 228
- "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.

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

Syntax

Defaults

-g0

Parameters

-g

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

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

-g1 Generates minimal read-only debugging information about line numbers and source file names. No program state is preserved. This option is equivalent to **-qlinedebug**.

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

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

-g3, -g4
Generates read-only debugging information about line numbers, source file names, and variables.

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

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

-g5, -g6, -g7
Generates read-only debugging information about line numbers, source file names, and variables.

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

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

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

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

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

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

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

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

Usage

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

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

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

- `if` constructs

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

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

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

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

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

```
xlc myprogram.c -O2 -g8
```

Related information

- "-fstandalone-debug" on page 99
- "-qlinedebug" on page 161
- "-qfullpath" on page 143
- "-O, -qoptimize" on page 76
- "-qkeepparm" on page 159

-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

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 8

-isystem (-qc_stdinc) (C only)

Category

Compiler customization

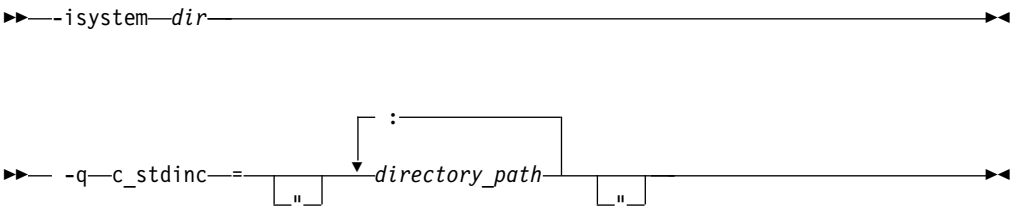
Pragma equivalent

None.

Purpose

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

Syntax



Defaults

By default, the compiler searches the directory specified in the configuration file for the XL C header files (this is normally /opt/ibm/xlC/13.1.3/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 8 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 118
- "-qstdinc, -qnostdinc (-nostdinc, -nostdinc++)" on page 198
- "-include (-qinclude)" on page 114
- "Directory search sequence for included files" on page 8
- "Specifying compiler options in a configuration file" on page 5
- "-I" on page 74

-isystem (-qcpp_stdinc) (C++ only)

Category

Compiler customization

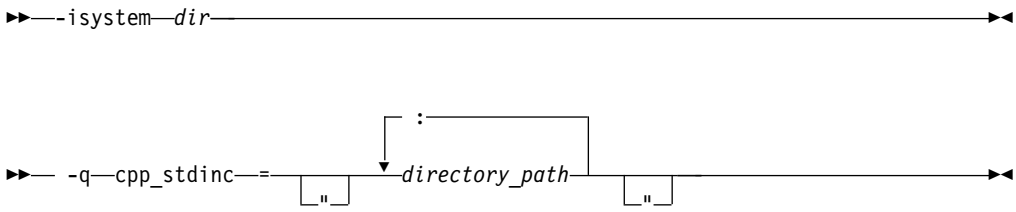
Pragma equivalent

None.

Purpose

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

Syntax



Defaults

By default, the compiler searches the directory specified in the configuration file for the XL C++ header files (this is normally /opt/ibm/xlC/13.1.3/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 8 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 119
- "-qstdinc, -qnostdinc (-nostdinc, -nostdinc++)" on page 198
- "-include (-qinclude)" on page 114
- "Directory search sequence for included files" on page 8
- "Specifying compiler options in a configuration file" on page 5
- "-I" on page 74

-isystem (-qgcc_c_stdinc) (C only)

Category

Compiler customization

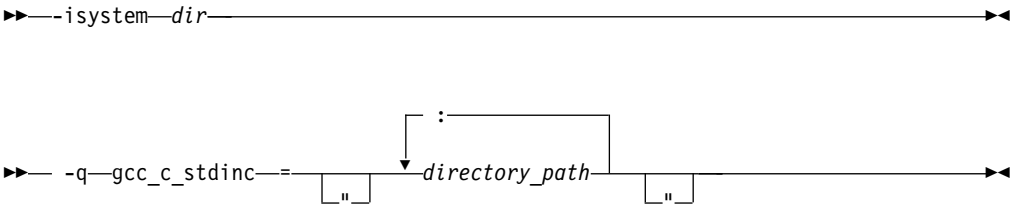
Pragma equivalent

None.

Purpose

Changes the standard search location for the GNU C system header files.

Syntax



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 8 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 115
- "-qstdinc, -qnostdinc (-nostdinc, -nostdinc++)" on page 198
- "-include (-qinclude)" on page 114
- "Directory search sequence for included files" on page 8
- "Specifying compiler options in a configuration file" on page 5
- "-I" on page 74

-isystem (-qgcc_cpp_stdinc) (C++ only)

Category

Compiler customization

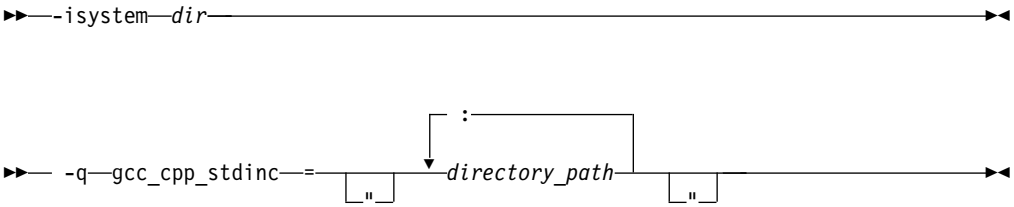
Pragma equivalent

None

Purpose

Changes the standard search location for the GNU C++ system header files.

Syntax



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 8 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 117
- "-qstdinc, -qnostdinc (-nostdinc, -nostdinc++)" on page 198
- "-include (-qinclude)" on page 114
- "Directory search sequence for included files" on page 8
- "Specifying compiler options in a configuration file" on page 5

- "-I" on page 74

-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

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 75
- "Specifying compiler options in a configuration file" on page 5

-maltivec (-qaltivec)

Category

Language element control

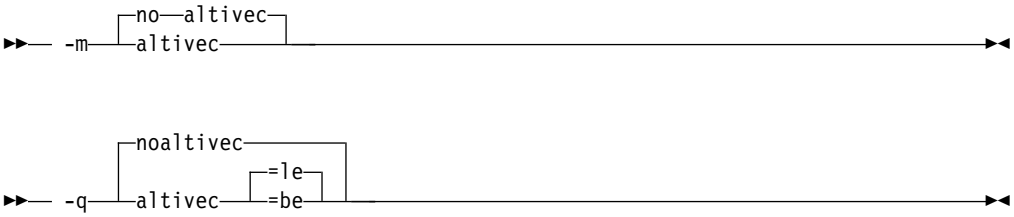
Pragma equivalent

None.

Purpose

Enables the compiler support for vector data types and operators.

Syntax



Defaults

By default, `-mno-altivec` or `-qnoaltivec` is effective. Specifying `-maltivec` is equivalent to specifying `-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` or `-qaltivec` option has effect only when you set or imply `-mcpu` to be an architecture that supports vector instructions. Otherwise, the compiler ignores `-maltivec` or `-qaltivec` and issues a warning message.

The `-maltivec` or `-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` or `-qaltivec` is in effect; otherwise, they are undefined.

__VEC_ELEMENT_REG_ORDER__ is defined to __ORDER_LITTLE_ENDIAN__ when -qaltivec=le (-maltivec) is in effect, or to __ORDER_BIG_ENDIAN__ when -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 -qaltivec=be
  ```

Related information

- "-mcpu (-qarch)"
- "Vector built-in functions" on page 309
- Vector types (IBM extension)
- "-qsimd" on page 190
- *AltiVec Technology Programming Interface Manual*, available at http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf

-mcpu (-qarch)

Category

Optimization and tuning

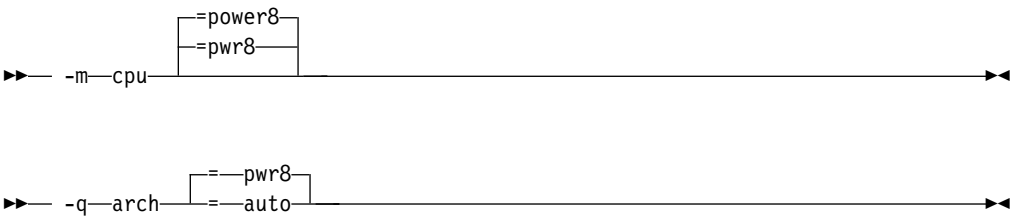
Pragma equivalent

None.

Purpose

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

Syntax



Defaults

- **-mcpu=pwr8**, **-mcpu=power8**, or **-qarch=pwr8**
- **-qarch=auto** when **-O4** or **-O5** is in effect

Parameters

auto
Automatically detects the specific architecture of the compilation machine. It assumes that the execution environment will be the same as the compilation environment. This option is implied if the **-O4** or **-O5** option is set or implied. You can specify the **auto** suboption with **-qarch** only.

pwr8
Produces object code containing instructions that run on the POWER8® hardware platforms.

power8
Produces object code containing instructions that run on the POWER8 hardware platforms. You can specify this suboption with **-march** only.

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)" on page 125.

The POWER8 architecture supports graphics, square root, Vector Multimedia Extension (VMX) processing, Vector Scalar Extension (VSX) processing, hardware transactional memory, and cryptography.

Predefined macros

See "Macros related to architecture settings" on page 269 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, enter:

```
xlc -o testing myprogram.c -mcpu=pwr8
```

Related information

- -qprefetch
- -qfloat
- “-mtune (-qtune)”
- “Macros related to architecture settings” on page 269
- "Optimizing your applications" in the *XL C/C++ Optimization and Programming Guide*

-mtune (-qtune)

Category

Optimization and tuning

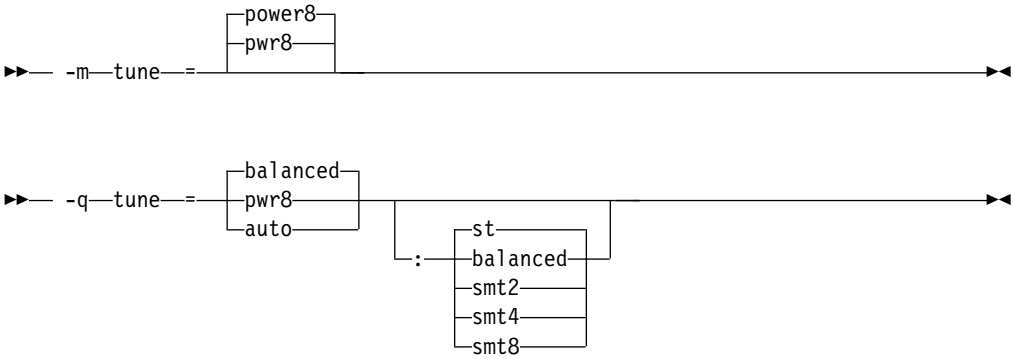
Pragma equivalent

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



Defaults

-mtune=pwr8 , **-mtune=power8**, or **-qtune=pwr8:st**

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.

pwr8
Optimizations are tuned for the POWER8 hardware platforms.

power8
Optimizations are tuned for the POWER8 hardware platforms. You can specify this suboption with **-mtune** only.

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.

Predefined macros

None.

Examples

To specify that the executable program `testing` compiled from `myprogram.c` is to be optimized for a POWER8 hardware platform, enter:

```
xlc -o testing myprogram.c -mtune=pwr8
```

To specify that the executable program `testing` compiled from `myprogram.c` is to be optimized for a POWER8 hardware platform configured for the SMT4 mode, enter:

```
xlc -o testing myprogram.c -qtune=pwr8:smt4
```

Related information

- "-mcpu (-qarch)" on page 123
- "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

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 86
- "-E" on page 71
- "-P" on page 79
- "-fsyntax-only (-qsyntaxonly)" on page 101

-p, -pg, -qprofile

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Prepares the object files produced by the compiler for profiling.

When you compile with a profiling option, the compiler produces monitoring code that counts the number of times each routine is called. The compiler replaces the startup routine of each subprogram with one that calls the monitor subroutine at the start. When you execute the compiled program and it ends normally, it writes the recorded information to a gmon.out file. You can then use the **gprof** command to generate a runtime profile.

Syntax

Defaults

Not applicable.

Usage

When you are compiling and linking in separate steps, you must specify the profiling option in both steps.

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

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

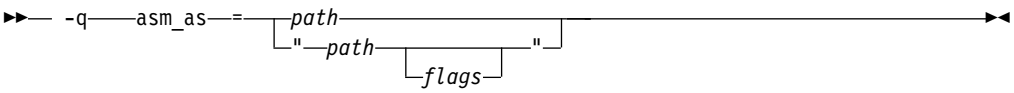
None.

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



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 88

-qcache

Category

Optimization and tuning

Pragma equivalent

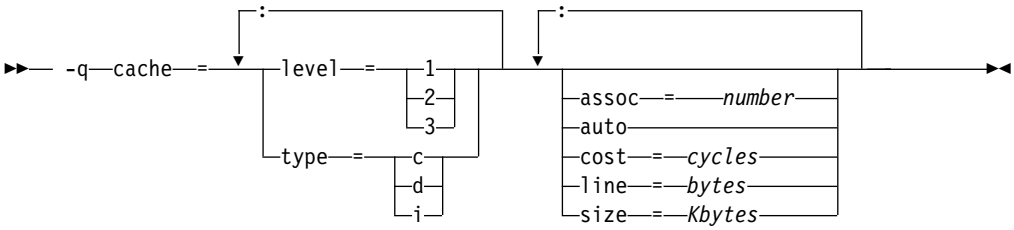
None.

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.

Syntax



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

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

level
Is one of:

1 Basic cache

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

3 TLB

line
Specifies the line size of the cache.

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

size
Specifies the total size of the cache.

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

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

cache_type
Is one of:

c Combined data and instruction cache

d Data cache

i Instruction cache

Usage

The **-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 **-O4** and **-O5** optimization options automatically select the cache characteristics of the compiling machine. If you specify the **-qcache** option together with the **-O4** or **-O5** options, the option specified last takes precedence.
- 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 -O4 -qcache=type=c:level=1:size=8:line=64:assoc=2 file.c
```

Related information

- "-qcache" on page 130
- "-O, -qoptimize" on page 76
- "-mtune (-qtune)" on page 125
- "-qipa" on page 152
- "Optimizing your applications" in the *XL C/C++ Optimization and Programming Guide*

-qcheck

Category

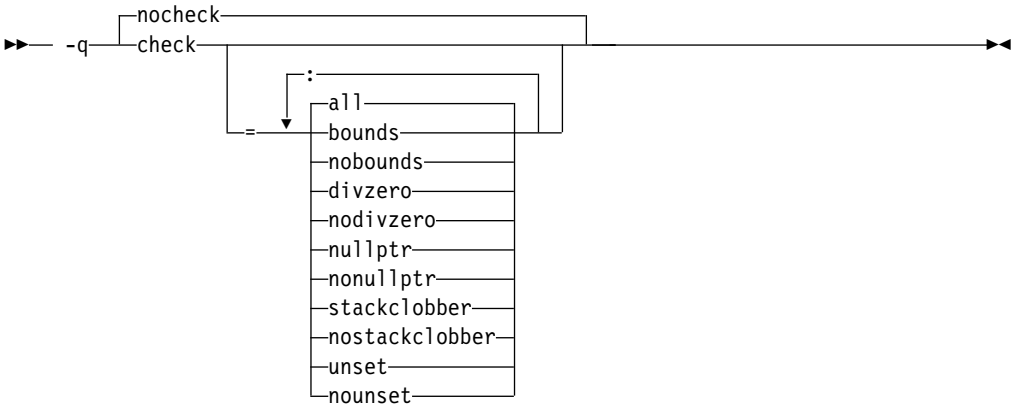
Error checking and debugging

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



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.

-qcompact

Category

Optimization and tuning

Purpose

Avoids optimizations that increase code size.

Syntax

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

Defaults

-qcrt

Predefined macros

None.

Related information

- "-qlib, -nodefaultlibs (-qnolib)" on page 159

-qdataimported, -qdatalocal, -qtocdata

Category

Optimization and tuning

Pragma equivalent

None.

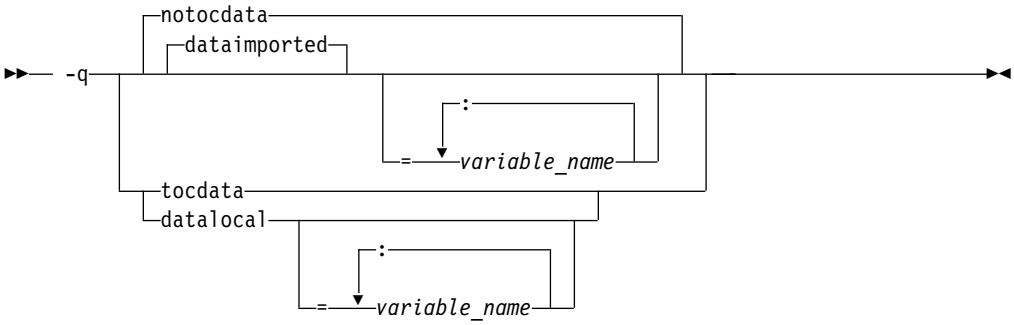
Purpose

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



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

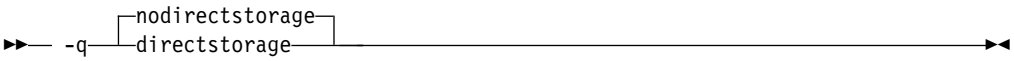
Pragma equivalent

None.

Purpose

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

Syntax



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.

-qeh (C++ only)

Category

Object code control

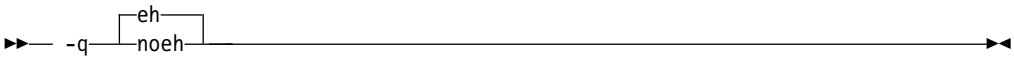
Pragma equivalent

None.

Purpose

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

Syntax



Defaults

-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.

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

Predefined macros

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

Related information

- "-qrtti, -fno-rtti (-qnortti) (C++ only)" on page 186
- The **-fexceptions** option that GCC provides. For details, see the GCC online documentation at http://gcc.gnu.org/onlinedocs/.

-qfloat

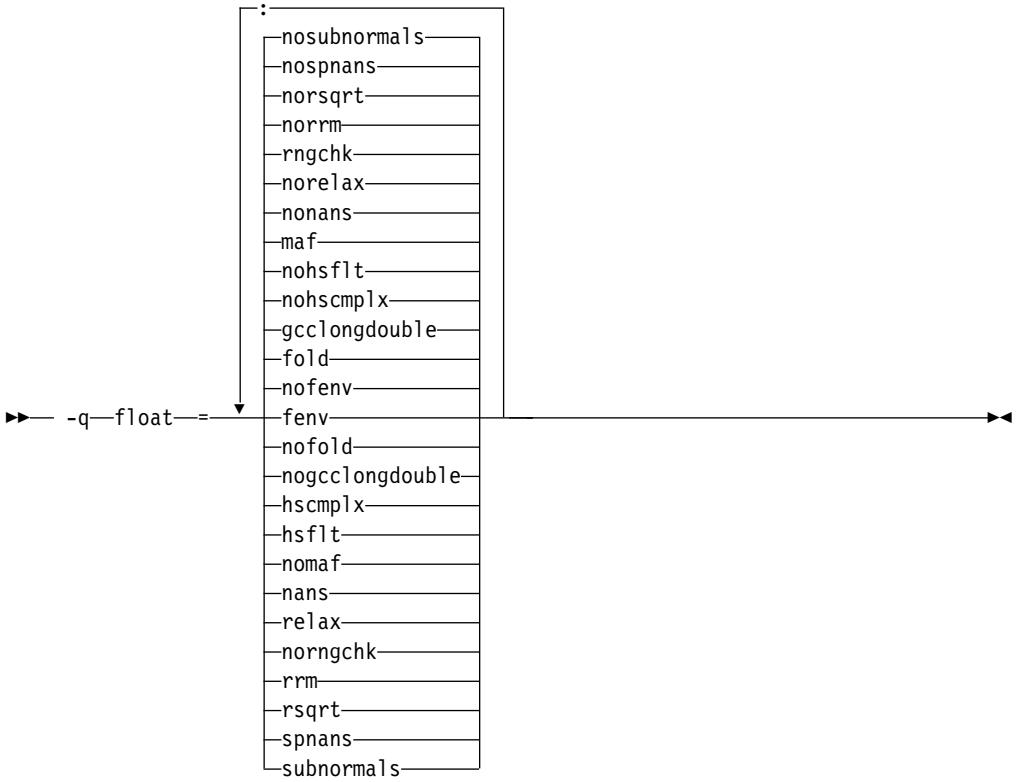
Category

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 **-O3** 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 281 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 | **nonans**
Allows you to use the **-qflttrap=invalid:enable** option to detect and deal with exception conditions that involve signaling NaN (not-a-number) values. Use this suboption only if your program explicitly creates signaling NaN values, because these values never result from other floating-point operations.

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

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

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

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

If any of these conditions are not met, incorrect results may be produced. For example, if the divisor for a division operation is 0.0 or a denormalized number (absolute value < 2^{-1022} for double precision, and absolute value < 2^{-126} for single precision), NaN, instead of INF, may result; when the divisor is +/- INF, NaN instead of 0.0 may result. If the input is +INF for a sqrt operation, NaN, rather than INF, may result.

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

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

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

rsqrt has no effect unless **-qignerrno** is also specified; `errno` will *not* be set for any `sqrt` function calls.

If you compile with the **-O3** or higher optimization level, **rsqrt** is enabled automatically. To disable it, also specify **-qstrict**, **-qstrict=nans**, **-qstrict=infinities**, **-qstrict=zerosigns**, or **-qstrict=exceptions**.

spnans | nospnans
Generates extra instructions to detect signalling NaN on conversion from single-precision to double-precision.

subnormals | nosubnormals
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 199.

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 123
- "-ftrapping-math (-qfltrap)" on page 103
- "-qstrict" on page 199
- "Handling floating-point operations" in the *XL C/C++ Optimization and Programming Guide*

-qfullpath

Category

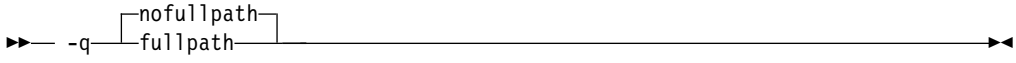
Error checking and debugging

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



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 161
- "-g" on page 112

-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

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

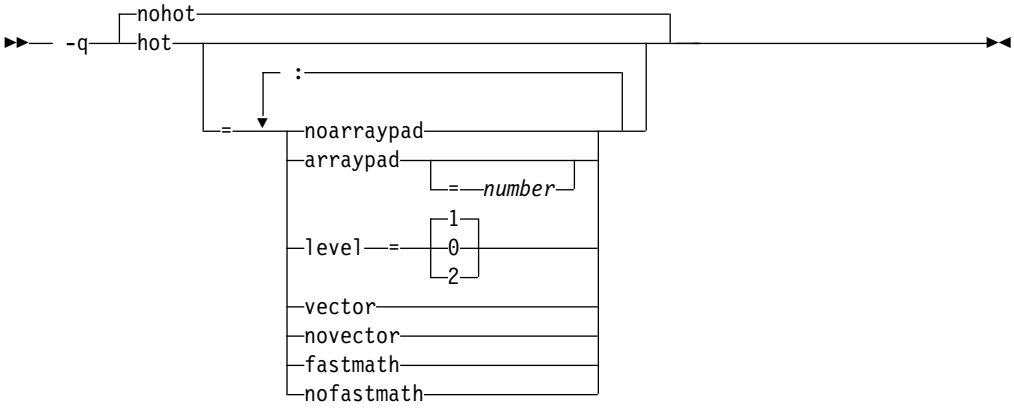
Optimization and tuning

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

Parameters

arraypad | noarraypad
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 **-O3 -qhot** or higher. For more information, see "-qreport" on page 180.

Predefined macros

None.

Related information

- "-mcpu (-qarch)" on page 123
- "-qsimd" on page 190
- "-qprefetch" on page 177
- "-qreport" on page 180
- "-qlistfmt" on page 163
- "-O, -qoptimize" on page 76
- "-qstrict" on page 199
- *Using the Mathematical Acceleration Subsystem (MASS)* in the *XL C/C++ Optimization and Programming Guide*
- "#pragma nosimd" on page 232

-qidirfirst

Category

Input control

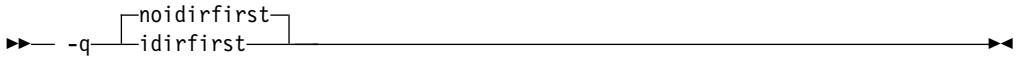
Pragma equivalent

None.

Purpose

Searches for user included files in directories that are specified by the **-I** option before searching any other directories.

Syntax



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 74
- "-include (-qinclude)" on page 114
- "-qstdinc, -qnostdinc (-nostdinc, -nostdinc++)" on page 198
- "-isystem (-qc_stdinc) (C only)" on page 115
- "-isystem (-qcpp_stdinc) (C++ only)" on page 117
- "Directory search sequence for included files" on page 8

-qignerrno

Category

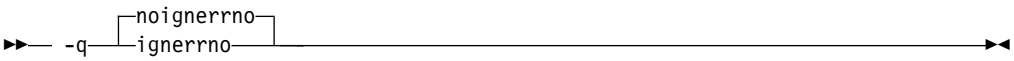
Optimization and tuning

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



Defaults

- -qnoignerrno
- **-qignerrno** when the **-O3** or higher optimization level is in effect.

Usage

If you require both **-O3** or higher and the ability to set `errno`, you should specify **-qnoignerrno** *after* the optimization option on the command line.

Predefined macros

C++ __IGNERRNO__ is defined to 1 when **-qignerrno** is in effect; otherwise, it is undefined.

Related information

- "-O, -qoptimize" on page 76

-qinitauto

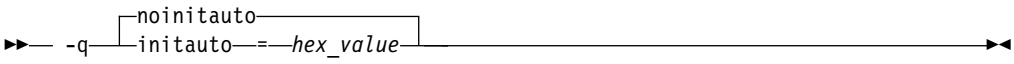
Category

Error checking and debugging

Purpose

Initializes uninitialized automatic variables to a specific value, for debugging purposes.

Syntax



Defaults

-qnoinitauto

Parameters

hex_value
A one- to eight-digit hexadecimal number.

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

- The *hex_value* can be optionally prefixed with `0x`, in which `x` is case-insensitive.

Usage

The -qinitauto option provides the following benefits:

- Setting *hex_value* to zero ensures that all automatic variables 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

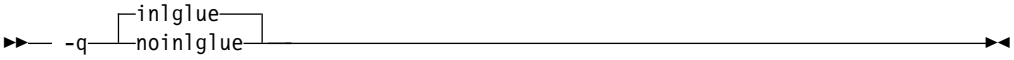
Object code control

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



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 135
- "-mtune (-qtune)" on page 125

-qipa

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

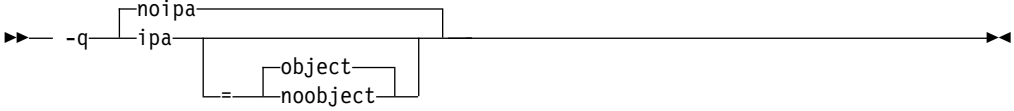
Enables or customizes a class of optimizations known as interprocedural analysis (IPA).

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

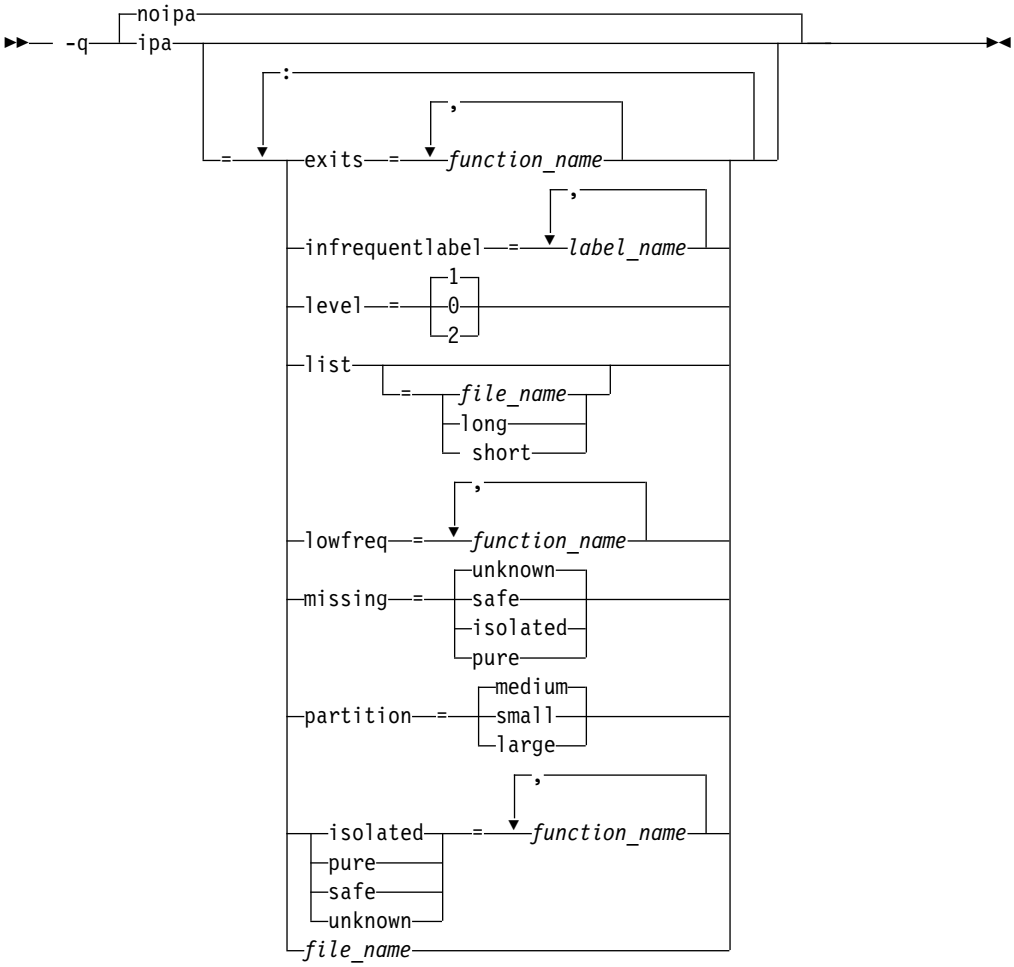
You can use **-qipa** during the compilation step, the link step, or both. If you compile and link in a single compiler invocation, only the link-time suboptions are relevant. If you compile and link in separate compiler invocations, only the compile-time suboptions are relevant during the compile step, and only the link-time suboptions are relevant during the link step.

Syntax

-qipa compile-time syntax



-qipa link-time syntax



Defaults

- **-qnoipa**

Parameters

You can specify the following parameters during a separate compile step only:

object | noobject
Specifies whether to include standard object code in the output object files.

Specifying **noobject** can substantially reduce overall compile time by not generating object code during the first IPA phase. Note that if you specify **-S** with **noobject**, **noobject** will be ignored.

If compiling and linking are performed in the same step and you do not specify the **-S** or any listing option, **-qipa=noobject** is implied.

Specifying **-qipa** with no suboptions on the compile step is equivalent to **-qipa=object**.

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

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:

short Requests less information in the listing file. Generates the Object File Map, Source File Map and Global Symbols Map sections of the listing.

long Requests more information in the listing file. Generates all of the sections generated by the **short** suboption, plus the Object Resolution Warnings, Object Reference Map, Inliner Report and Partition Map sections.

lowfreq
Specifies functions that are likely to be called infrequently. These are typically error handling, trace, or initialization functions. The compiler may be able to make other parts of the program run faster by doing less optimization for calls to these functions.

missing
Specifies the interprocedural behavior of functions that are not compiled with **-qipa** and are not explicitly named in an **unknown**, **safe**, **isolated**, or **pure** suboption.

Valid suboptions are one of the following suboptions:

safe Specifies that the missing functions do not indirectly call a visible (not missing) function either through direct call or through a function pointer.

isolated
Specifies that the missing functions do not directly reference global variables accessible to visible function. Functions bound from shared libraries are assumed to be *isolated*.

pure Specifies that the missing functions are *safe* and *isolated* and do not indirectly alter storage accessible to visible functions. *pure* functions also have no observable internal state.

unknown
Specifies that the missing functions are not known to be *safe*, *isolated*, or *pure*. This suboption greatly restricts the amount of interprocedural optimization for calls to missing functions.

The default is to assume **unknown**.

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

- **small**
- **medium**
- **large**

Larger partitions contain more functions, which result in better interprocedural analysis but require more storage to optimize. Reduce the partition size if compilation takes too long because of paging.

pure
Specifies *pure* functions that are not compiled with **-qipa**. Any function

specified as *pure* must be *isolated* and *safe*, and must not alter the internal state nor have side-effects, defined as potentially altering any data visible to the caller.

safe
Specifies *safe* functions that are not compiled with **-qipa** and do not call any other part of the program. Safe functions can modify global variables, but may not call functions compiled with **-qipa**.

unknown
Specifies *unknown* functions that are not compiled with **-qipa**. Any function specified as *unknown* can make calls to other parts of the program compiled with **-qipa**, and modify global variables.

file_name
Gives the name of a file which contains suboption information in a special format.

The file format is shown as follows:

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

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

Usage

Specifying **-qipa** automatically sets the optimization level to **-O2**. For additional performance benefits, you can also specify the **-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 93
- "-qisolated_call"
- "#pragma execution_frequency" on page 230
- "-S" on page 81
- "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.

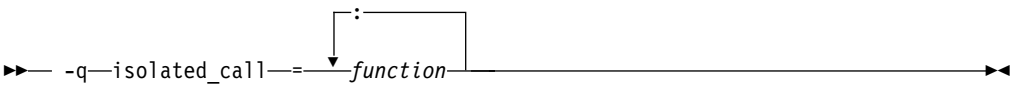
Essentially, any change in the state of the runtime environment is considered a side effect, including:

- 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



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

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 76

-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

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 136

-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

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.

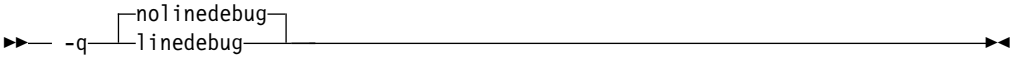
Purpose

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



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 112
- "-O, -qoptimize" on page 76

-qlist

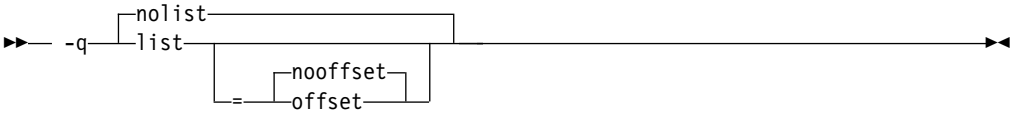
Category

Listings, messages, and compiler information

Purpose

Produces a compiler listing file that includes object and constant area sections.

Syntax



Defaults

-qnolist

Parameters

offset | nooffset

Changes the offset of the PDEF header from `00000` to the offset of the start of the text area. Specifying the option allows any program reading the .lst file to add the value of the PDEF and the line in question, and come up with the same value whether **offset** or **nooffset** is specified. The **offset** suboption is only relevant if there are multiple procedures in a compilation unit.

Specifying **list** without the suboption is equivalent to **list=nooffset**.

Usage

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

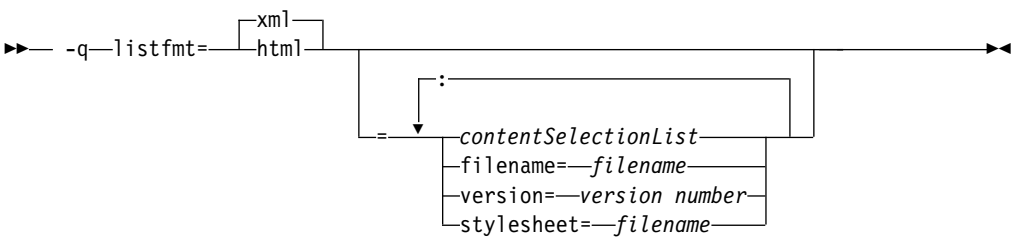
Pragma equivalent

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:

xml | **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 command" on page 166.

contentSelectionList
The following suboptions provide a filter to limit the type and quantity of information in the report:

data | **nodata**
Produces data reorganization information.

inlines | **noinlines**
Produces inlining information.

pdf | **nopdf**
Produces profile-directed feedback information.

transforms | **notransforms**
Produces loop transformation information.

all
Produces all available report information.

none
Does not produce a report.

filename
Specifies the name of the report file. One file is produced during the compile phase, and one file is produced during the IPA link phase. If no filename is specified, a file with the suffix `.xml` or `.html` is generated in a way that is consistent with the rules of name generation for the given platform. For example, if 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/13.1.3/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, V13.1.3 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
```

`genhtml` command

To view the HTML version of an XML report that has already been generated, you can use the **`genhtml`** tool.

Use the following command to view the existing XML report in HTML format. This command generates the HTML content to standard output.

```
genhtml xml_file
```

Use the following command to generate the HTML content into a defined HTML file. You can use a web browser to view the generated HTML file.

```
genhtml xml_file > target_html_file
```

Note: The 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

- "-qreport" on page 180
- "Using compiler reports to diagnose optimization opportunities" in the *XL C/C++ Optimization and Programming Guide*

-qmaxmem

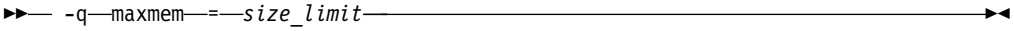
Category

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



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

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 8. 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 127
- "Directory search sequence for included files" on page 8
- 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/.

-qpath

Category

Compiler customization

Pragma equivalent

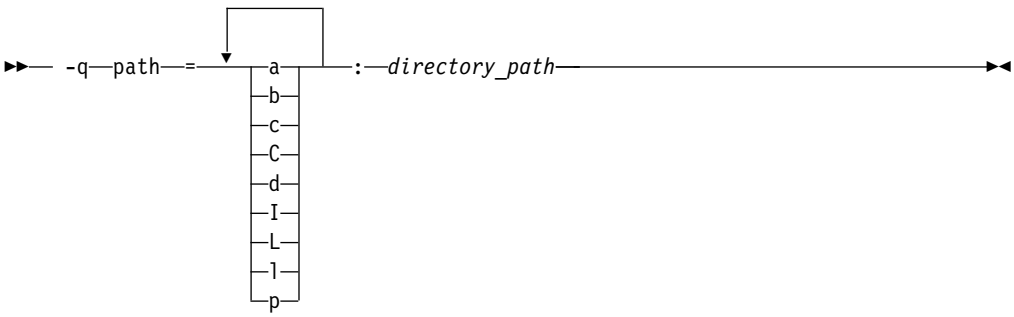
None.

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
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
p	The preprocessor	xlCentry

Usage

The **-qpath** option overrides the **-F**, **-t**, and **-B** options.

Predefined macros

None.

Examples

To compile `myprogram.c` using a substitute **xlc** compiler 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 68
- "-F" on page 72
- "-t" on page 216

-qpdf1, -qpdf2

Category

Optimization and tuning

Pragma equivalent

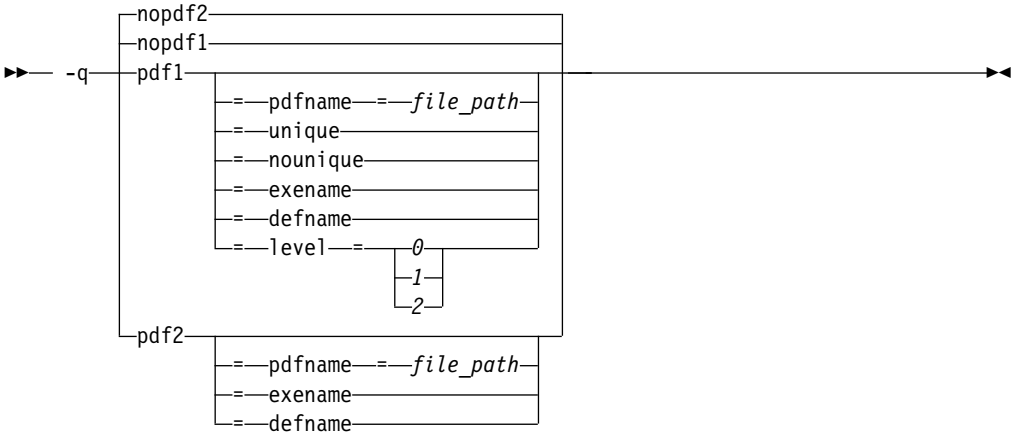
None.

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, -qnopdf2

Parameters

defname
Reverts a PDF file to its default file name if the **-qpdf1=exename** option is also specified.

exename
Specifies the name of the generated PDF file according to the output file name specified by the **-o** option. For example, you can use **-qpdf1=exename -o func func.c** to generate a PDF file called `.func_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 21. Profiling type supported on each -qpdf1 level

	Level		
Profiling type	**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 compiled with the **-qpdf1=level=2** option can be run at a time on a particular processor.
- Cache-miss profiling information has several levels. 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) accordingly. 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.

pdfname= *file_path*

Specifies the directories and names for the PDF files and any existing PDF map files. By default, if the PDFDIR environment variable is set, the compiler places the PDF and PDF map files in the directory specified by PDFDIR. Otherwise, if the PDFDIR environment variable is not set, 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 using the same directory. This is especially useful when you are tuning dynamic libraries with PDF.

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 `._pdf` by default or specified by other **-qpdf1** suboptions, which include **pdfname**, **exename**, and **defname**. *<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 named `._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 `._pdf` by default. This step is called the PDF training step.

3. Recompile and link or just 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.

Notes:

- The **showpdf** utility uses the PDF map file to display part of the profiling information in text or XML format. For details, see "Viewing profiling information with 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 **-qpdf1** phase so that the PDF map file is not generated. For details of **-qnoshowpdf**, see **-qshowpdf** in the *XL C/C++ Compiler Reference*.
- When option **-O4**, **-O5**, or any level of option **-qipa** is in effect, and you specify the **-qpdf1** or **-qpdf2** option at the link step but not at the compile step, the compiler issues a warning message. The message indicates that you must recompile your program to get all the profiling information.
- When the **-qpdf1=pdfname** option is used during the **-qpdf1** phase, you must use the **-qpdf2=pdfname** option during the **-qpdf2** phase for the compiler to recognize the correct PDF file. This rule also applies to the **-qpdf[1|2]=exename** option.

The compiler issues an information message with a number in the range of 0 - 100 during the **-qpdf2** phase. If you have not changed your program between the **-qpdf1** and **-qpdf2** phases, the number is 100, which means that all the profiling information can be used to optimize the program. If the number is 0, it means that the profiling information is completely outdated, and the compiler cannot take advantage of any information. When the number is less than 100, you can choose to recompile your program with the **-qpdf1** option and regenerate the profiling information.

If you recompile your program by using the **-qpdf1** option with any suboption, the compiler removes the existing PDF file or files whose names and locations are the same as the file or files that will be created in the training step before generating a new application.

Other related options

You can use the following option with the **-qpdf1** option:

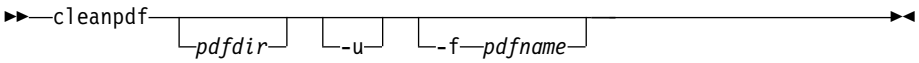
-qprefetch
When you run the **-qprefetch=assistthread** option to generate data prefetching assist threads, the compiler uses the delinquent load information to perform analysis and generate them. The delinquent load information can be gathered from dynamic profiling using the **-qpdf1=level=2** option. For more information, see -qprefetch.

-qshowpdf
Uses the **showpdf** utility to view the PDF data that were collected. See "-qshowpdf" on page 189 for more information.

For recommended procedures of using PDF, see "Using profile-directed feedback" in the *XL C/C++ Optimization and Programming Guide*.

The following utility programs, found in `/opt/ibm/xlC/13.1.3/bin/`, are available for managing the files to which profiling information is written:

cleanpdf



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.

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, `._pdf` is removed.

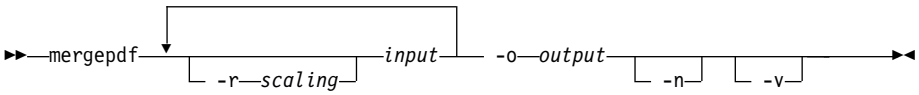
-u If `-f` *pdfname* is specified, in addition to the file removed by **-f**, files with the naming convention *pdfname*`.<`*pid*`>`, if applicable, are also removed.

If `-f` *pdfname* is not specified, removes `._pdf`. Files with the naming convention `._pdf.<`*pid*`>`, if applicable, are also removed.

<*pid*> is the ID of the running process in the PDF training step.

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**.

mergepdf



Merges two or more PDF files into a single PDF file.

-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. 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 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.

-v Specifies verbose mode, and causes internal and user-specified scaling ratios to be displayed to standard output.

showpdf

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. See "Viewing profiling information with showpdf" in the *XL C/C++ Optimization and Programming Guide* for more information.

Predefined macros

None.

Examples

The following example uses the **-qpdf1=level=0** option to reduce possible runtime instrumentation overhead:

```
#Compile all the files with -qpdf1=level=0
xlc -qpdf1=level=0 -O3 file1.c file2.c file3.c

#Run with one set of input data
./a.out < sample.data

#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
```

The following example uses the **-qpdf1=level=1** option:

```
#Compile all the files with -qpdf1
xlc -qpdf1 -O3 file1.c file2.c file3.c

#Run with one set of input data
./a.out < sample.data

#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
```

The following example uses the **-qpdf1=level=2** option to gather cache-miss profiling information:

```
#Compile all the files with -qpdf1=level=2
xlc -qpdf1=level=2 -O3 file1.c file2.c file3.c

#Set PM_EVENT=L2MISS to gather L2 cache-miss profiling
#information
export PDF_PM_EVENT=L2MISS

#Run with one set of input data
./a.out < sample.data

#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
```

The following example demonstrates the use of the PDF_BIND_PROCESSOR environment variable:

```
#Compile all the files with -qpdf1=level=1
xlc -qpdf1=level=1 -O3 file1.c file2.c file3.c
```

```
#Set PDF_BIND_PROCESSOR environment variable so that
#all processes for this executable are run on Processor 1
export PDF_BIND_PROCESSOR=1

#Run executable with sample input data
./a.out < sample.data

#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
```

The following example demonstrates the use of the **-qpdf[1|2]=exename** option:

```
#Compile all the files with -qpdf1=exename
xlc -qpdf1=exename -O3 -o final file1.c file2.c file3.c

#Run executable with sample input data
./final < typical.data

#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

#Recompile all the files with -qpdf2=exename
xlc -qpdf2=exename -O3 -o final file1.c file2.c file3.c

#The program is now optimized using PDF information
```

The following example demonstrates the use of the **-qpdf[1|2]=pdfname** option:

```
#Compile all the files with -qpdf1=pdfname. The static profiling
#information is recorded in a file named final_map
xlc -qpdf1=pdfname=final -O3 file1.c file2.c file3.c

#Run executable with sample input data. The profiling
#information is recorded in a file named final
./a.out < typical.data

#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

#Recompile all the files with -qpdf2=pdfname
xlc -qpdf2=pdfname=final -O3 file1.c file2.c file3.c

#The program is now optimized using PDF information
```

Related information

- "-qshowpdf" on page 189
- "-qipa" on page 152
- -qprefetch
- "-qreport" on page 180
- "Optimizing your applications" in the *XL C/C++ Optimization and Programming Guide*

- "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

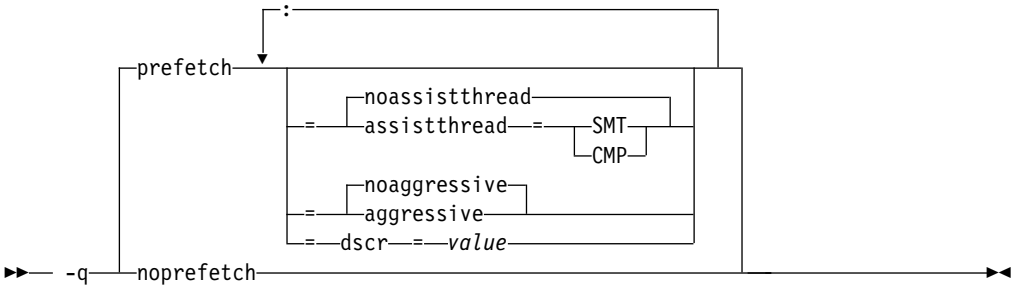
None.

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



Defaults

-qprefetch=noassistthread:noaggressive:dscr=0

Parameters

assistthread | noassistthread
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 | noaggressive
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 only 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

- -march (-qarch)
- "-qhot" on page 145
- "-qpdf1, -qpdf2" on page 170
- "-qreport" on page 180
- "__mem_delay" on page 446

-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

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

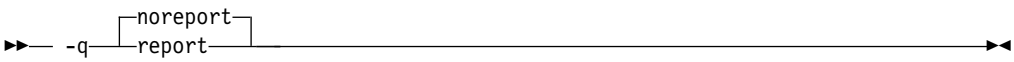
Produces listing files that show how sections of code have been optimized.

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 processors support both load and store stream prefetch.

Syntax



Defaults

-qnoreport

Usage

To generate a loop transformation listing, you must specify **-qreport** with one of the following options:

- **-qhot**
- **-qsmp**
- **-O3** or higher

To generate PDF information in the listing, you must specify both **-qreport** and **-qpdf2**.

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:

```
xlc -qhot -O3 -qreport myprogram.c
```

Related information

- "-qhot" on page 145
- "-qsimd" on page 190
- "-qipa" on page 152

-qreserved_reg

Category

Object code control

Pragma equivalent

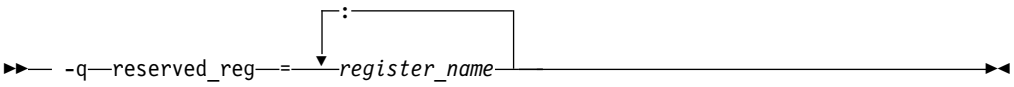
None.

Purpose

Indicates that the given list of registers cannot be used during the compilation except as a stack pointer, frame pointer or in some other fixed role.

You should use this option in modules that are required to work with other modules that use global register variables or hand-written assembler code.

Syntax



Defaults

Not applicable.

Parameters

register_name
A valid register name on the target platform. Valid registers are:

r0 to r31
General purpose registers

f0 to f31
Floating-point registers

v0 to v31
Vector registers (on selected processors only)

Usage

-qreserved_reg is cumulative, for example, specifying `-qreserved_reg=r14` and `-qreserved_reg=r15` is equivalent to specifying `-qreserved_reg=r14:r15`.

Duplicate register names are ignored.

Predefined macros

None.

Examples

To specify that `myprogram.c` reserves the general purpose registers r3 and r4, enter:

```
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

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 99

-qro

Category

Object code control

Purpose

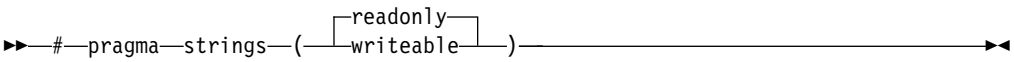
Specifies the storage type for string literals.

When **ro** or **strings=readonly** is in effect, strings are placed in read-only storage. When **noro** or **strings=writeable** is in effect, strings are placed in read/write storage.

Syntax

Option syntax

Pragma syntax



Defaults

C Strings are read-only for all invocation commands except **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 184
- "-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

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 184

-qrtti, -fno-rtti (-qnortti) (C++ only)

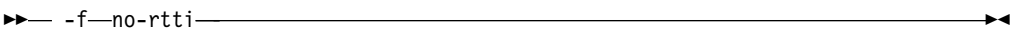
Category

Object code control

Purpose

Generates runtime type identification (RTTI) information for exception handling and for use by the `typeid` and `dynamic_cast` operators.

Syntax



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

- "-qeh (C++ only)" on page 139

-qsaveopt

Category

Object code control

Pragma equivalent

None.

Purpose

Saves the command-line options used for compiling a source file, the user's configuration file name and the options specified in the configuration files, the version and level of each compiler component invoked during compilation, and other information to the corresponding object file.

Syntax

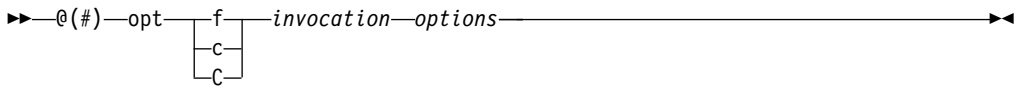
Defaults

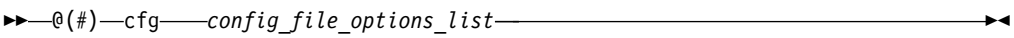
-qnosaveopt

Usage

This option has effect only when compiling to an object (.o) file (that is, using the **-c** option). Though each object might contain multiple compilation units, only one copy of the command-line options is saved. Compiler options specified with pragma directives are ignored.

Command-line compiler options information is copied as a string into the object file, using the following format:





where:

f Signifies a Fortran language compilation.

c Signifies a C language compilation.

C Signifies a C++ language compilation.

invocation
Shows the command used for the compilation, for example, **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.

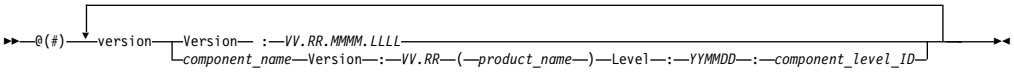
env_var_definition
The environment variables that are used by the compiler. Currently only **XLC_USR_CONFIG** is listed.

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:



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 13.1.3.0
@(#)opt c /opt/ibm/xlC/13.1.3/bin/xlC \
-F/opt/ibm/xlC/13.1.3/etc/xlc.cfg.rhel.7.1.gcc.4.8.3 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, V13.1.3 (5725-C73, 5765-J08)
@(#)version Version: 13.01.0003.0000
@(#)version Driver Version: 13.1.3(C/C++) Level: 151105 ID: _JbNFoYQ_EeWg_O7EssfHAg
@(#)version C/C++ Front End Version: 13.1.3(C/C++) Level: 151106 ID: _JX7IIIQ_EeWg_O7EssfHAg
@(#)version High-Level Optimizer Version: 13.1.3(C/C++) and 15.1.3(Fortran) Level: 151106
ID: _JfAAgYQ_EeWg_O7EssfHAg
@(#)version Low-Level Optimizer Version: 13.1.3(C/C++) and 15.1.3(Fortran) Level: 151030
ID: _sk208X8mEeWg_O7EssfHAg
```

In the first line, `c` identifies the source used as C, `/opt/ibm/xlC/13.1.3/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 64

-qshowpdf

Category

Optimization and tuning

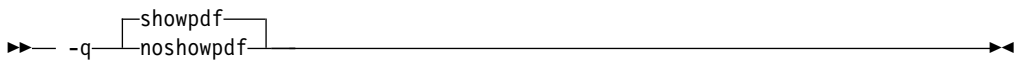
Pragma equivalent

None.

Purpose

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.

Syntax



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, the file is named `._pdf`).

In addition to the PDF file, the compiler also generates a PDF map file that contains static information during the **-qpdf1** phase. 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 "Viewing profiling information with 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 **-qpdf1** phase so that the PDF map file is not generated. This can reduce your compile time.

Predefined macros

None.

Related information

- "-qpdf1, -qpdf2" on page 170
- "Optimizing your applications" 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

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 199.

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 232
- "-mcpu (-qarch)" on page 123
- "-qreport" on page 180
- "-qstrict" on page 199
- *Using interprocedural analysis* in the *XL C/C++ Optimization and Programming Guide*.

-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

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**, **-O4**, or **-O5** 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 112
- "-qipa" on page 152

- "-O, -qoptimize" on page 76

-qsmp

Category

Optimization and tuning

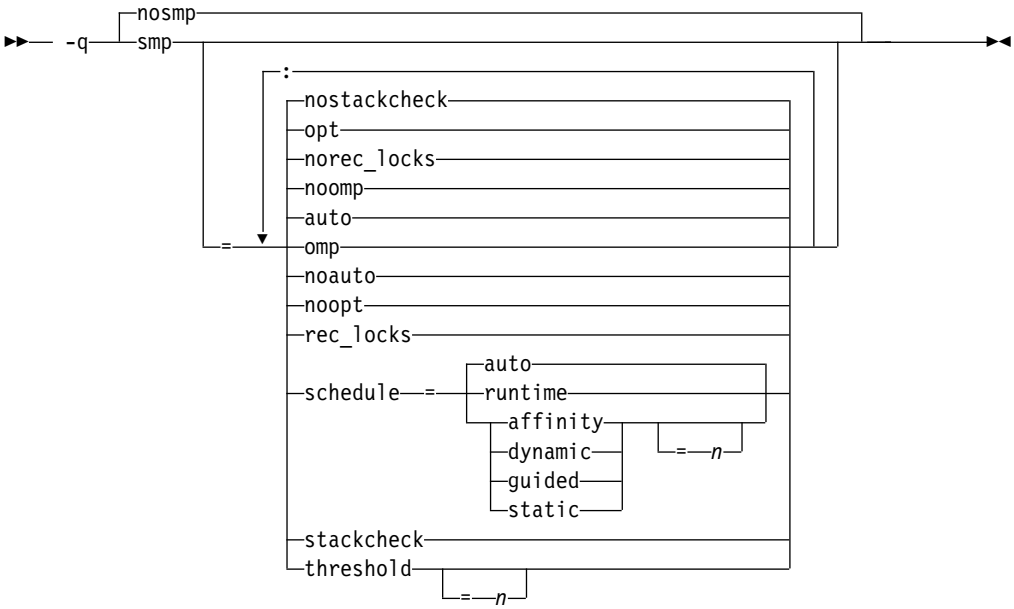
Pragma equivalent

None.

Purpose

Enables parallelization of program code.

Syntax



Defaults

-qnosmp. Code is produced for a uniprocessor machine.

Parameters

auto | **noauto**
Enables or disables automatic parallelization and optimization of program code. When **noauto** is in effect, only program code explicitly parallelized with OpenMP directives is optimized. **noauto** is implied if you specify **-qsmp=omp** or **-qsmp=noopt**.

omp | **noomp**
Enforces or relaxes strict compliance with the OpenMP standard. When **noomp** is in effect, **auto** is implied. When **omp** is in effect, **noauto** is implied and only OpenMP parallelization directives are recognized. The compiler issues warning messages if your code contains any language constructs that do not conform to the OpenMP API.

Note: The **-qsmp=omp** option must be used to enable OpenMP parallelization.

opt | noopt
Enables or disables optimization of parallelized program code. When **noopt** is in effect, the compiler will do the smallest amount of optimization that is required to parallelize the code. This is useful for debugging because **-qsmp** enables the **-O2** and **-qhot** options by default, which may result in the movement of some variables into registers that are inaccessible to the debugger. However, if the **-qsmp=noopt** and **-g** options are specified, these variables will remain visible to the debugger.

rec_locks | norec_locks
Determines whether recursive locks are used. When **rec_locks** is in effect, nested critical sections will not cause a deadlock. Note that the **rec_locks** suboption specifies behavior for critical constructs that is inconsistent with the OpenMP API.

schedule
Specifies the type of scheduling algorithms and, except in the case of **auto**, chunk size (*n*) that are used for loops to which no other scheduling algorithm has been explicitly assigned in the source code. Suboptions of the **schedule** suboption are as follows:

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 | nostackcheck
Causes the compiler to check for stack overflow by slave threads at run time, and issue a warning if the remaining stack size is less than the number of bytes specified by the **stackcheck** option of the XLSMPOPTS environment variable. This suboption is intended for debugging purposes, and only takes effect when **XLSMPOPTS=stackcheck** is also set; see “XLSMPOPTS” on page 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 **-O2**. The **-qsmp** option overrides **-qnooptimize**, but does not override **-O3**, **-O4**, or **-O5**. When debugging parallelized program code, you can disable optimization in parallelized program code by specifying **-qsmp=noopt**.
- The **-qsmp=noopt** suboption overrides performance optimization options anywhere on the command line unless **-qsmp** appears after **-qsmp=noopt**. For example, **-qsmp=noopt -O3** is equivalent to **-qsmp=noopt**, while **-qsmp=noopt -O3 -qsmp** is equivalent to **-qsmp -O3**.

Related information

- "-O, -qoptimize" on page 76

-qspill

Category

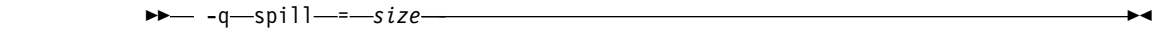
Compiler customization

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



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

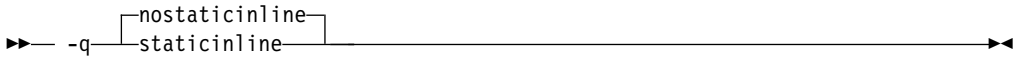
None.

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



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.

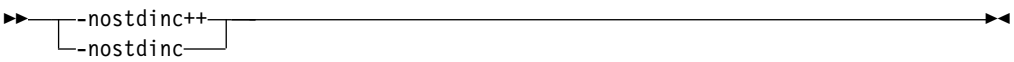
When **-qstdinc** is in effect, the compiler searches the following directories for header files:

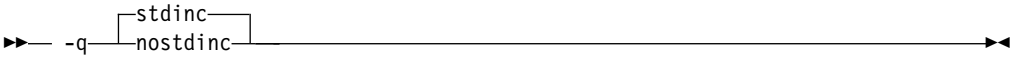
- C The directory specified in the configuration file for the XL C header files (this is normally /opt/ibm/xlC/13.1.3/include/) or by the **-isystem (-qc_stdinc)** option
- C++ The directory specified in the configuration file for the XL C and C++ header files (this is normally /opt/ibm/xlC/13.1.3/include/) 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





Defaults

-qstdinc

Usage

The search order of header files is described in “Directory search sequence for included files” on page 8.

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 115
- "-isystem (-qcpp_stdinc) (C++ only)" on page 117
- "-isystem (-qgcc_c_stdinc) (C only)" on page 118
- "-isystem (-qgcc_cpp_stdinc) (C++ only)" on page 119
- "-I" on page 74
- "Directory search sequence for included files" on page 8

-qstrict

Category

Optimization and tuning

Pragma equivalent

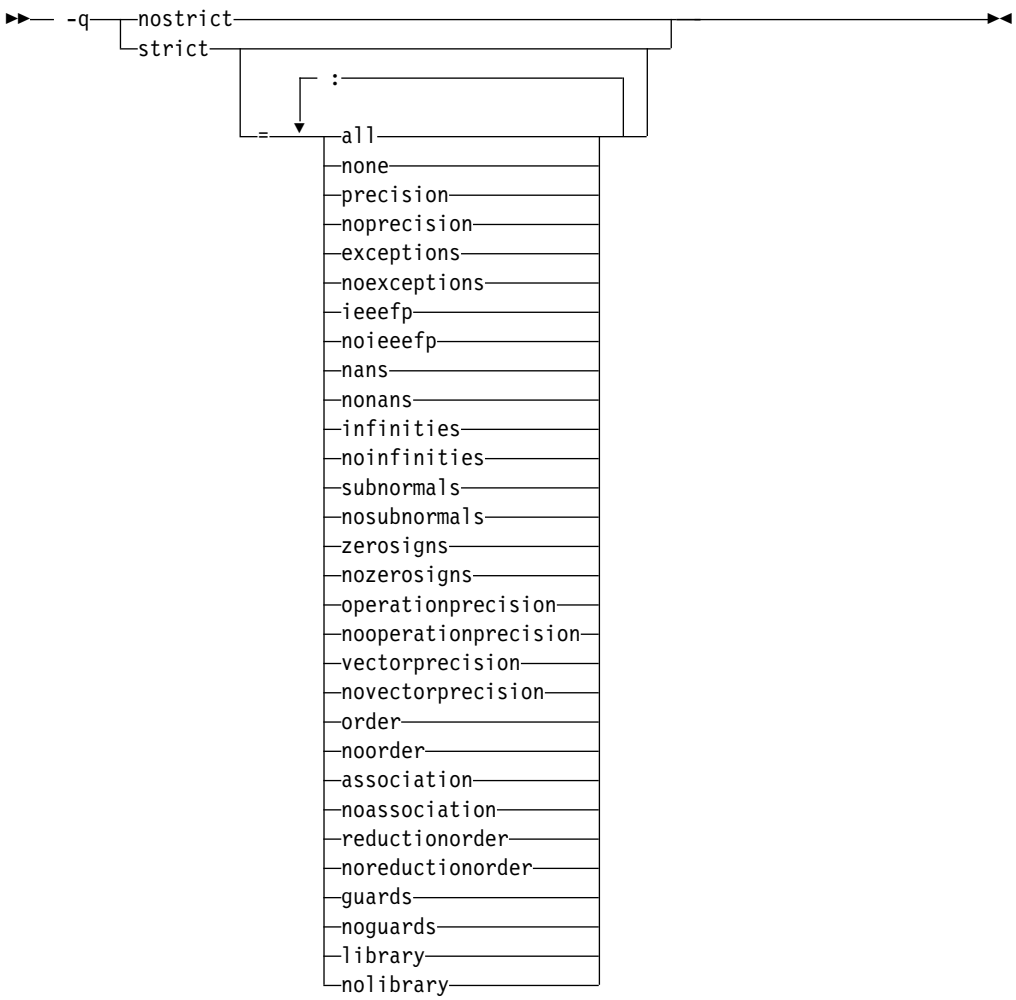
None.

Purpose

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.

This option is intended for situations where the changes in program execution in optimized programs produce different results from unoptimized programs.

Syntax



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**, **-qfloat=nomaf**, or **-qfloat=strictnmaf**.)
- Floating-point expressions may be reassociated. For example, **(2.0*3.1)*4.2** might become **2.0*(3.1*4.2)** if that is faster, even though the result might not be identical.
- The 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 139.

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 190
- "-qfloat" on page 139
- "-qhot" on page 145
- "-O, -qoptimize" on page 76

-qstrict_induction

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Prevents the compiler from performing induction (loop counter) variable optimizations. These optimizations may be unsafe (may alter the semantics of your program) when there are integer overflow operations involving the induction variables.

Syntax

Defaults

- **-qstrict_induction**
- **-qnostrict_induction** when **-O2** or higher optimization level is in effect

Usage

When using **-O2** or higher optimization, you can specify **-qstrict_induction** to prevent optimizations that change the result of a program if truncation or sign extension of a loop induction variable should occur as a result of variable overflow or wrap-around. However, use of **-qstrict_induction** is generally not recommended because it can cause considerable performance degradation.

Predefined macros

None.

Related information

- "-O, -qoptimize" on page 76

-qtimestamps

Category

"Output control" on page 47

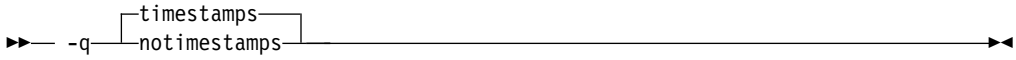
Pragma equivalent

None.

Purpose

Controls whether or not implicit time stamps are inserted into an object file.

Syntax



Defaults

-qtimestamps

Usage

By default, the compiler inserts an implicit time stamp in an object file when it is created. In some cases, comparison tools may not process the information in such binaries properly. Controlling time stamp generation provides a way of avoiding such problems. To omit the time stamp, use the option **-qnotimestamps**.

This option does not affect time stamps inserted by pragmas and other explicit mechanisms.

-qtmplinst (C++ only)

Category

Template control

Pragma equivalent

None.

Purpose

Manages the implicit instantiation of templates.

Syntax

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" on page 59

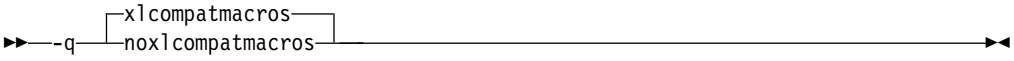
Pragma equivalent

None

Purpose

Defines the following legacy macros: C++ __IBMCPP__, __xlC__, __xlC_ver__ C++ , C __IBMC__, and __xlc__ C . This option helps you migrate programs from IBM XL C/C++ for Linux for big endian distributions to IBM XL C/C++ for Linux V13.1.3 for little endian distributions.

Syntax



Defaults

-qxlcompatmacros

Usage

The **-qxlcompatmacros** option is enabled by default to help you migrate programs from Linux for big endian distributions to Linux for little endian distributions. This means that the compiler predefines C++ __IBMCPP__, __xlC__, __xlC_ver__ C++ , C __IBMC__, and __xlc__ C .

When you migrate programs from V13.1.1 Linux for little endian distributions to V13.1.3 Linux for little endian distributions, it is recommended that you use the **-qnoxlcompatmacros** option to undefine these legacy macros. This is because these legacy macros, if defined, might change your source code and result in compilation failure.

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
- C++ __xlC__ C++
- C++ __xlC_ver__ C++

Related information

"Macros indicating the XL C/C++ compiler" on page 264

"-D" on page 70

-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

Defaults

-qunwind

Usage

The `setjmp` and `longjmp` families of library functions are safe to use with **-qnounwind**.

C++ Specifying **-qnounwind** also implies **-qnoeh**.

Predefined macros

None.

Related information

- "-qeh (C++ only)" on page 139

-r

Category

Object code control

Pragma equivalent

None.

Purpose

Produces a nonexecutable output file to use as an input file in another ld command call. This file may also contain unresolved symbols.

Syntax

Defaults

Not applicable.

Usage

A file produced with this flag is expected to be used as an input file in another compiler invocation or ld command call.

Predefined macros

None.

Examples

To compile `myprogram.c` and `myprog2.c` into a single object file `mytest.o`, enter:

```
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

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 macks

None.

Related information

- "-g" on page 112

-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

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 88
- "-qipa" on page 152
- "-o" on page 127
- "-fPIC (-qpic)" on page 96
- "-qpriority (C++ only)" on page 179
- "-fvisibility (-qvisibility)" on page 110
- "Supported GCC pragmas" on page 228
- "-static (-qstaticlink)"

-static (-qstaticlink)

Category

Linking

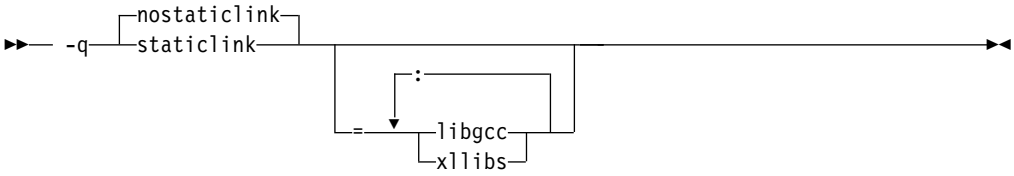
Pragma equivalent

None.

Purpose

Controls whether static or shared runtime libraries are linked into an application.

Syntax



The following table shows the equivalent usage between different format of options for specifying the linkage of shared and nonshared libraries.

Table 22. 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 23. 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**	
-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 209

-std (-qlanglvl)

Category

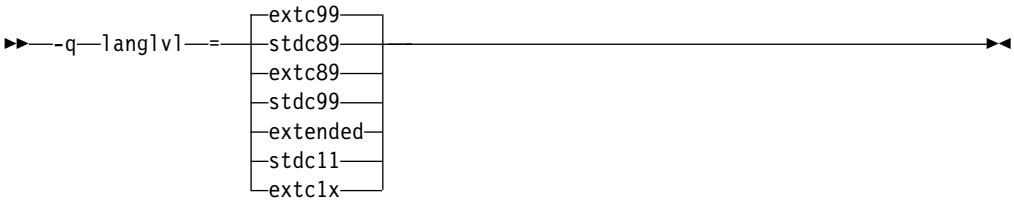
Language element control

Purpose

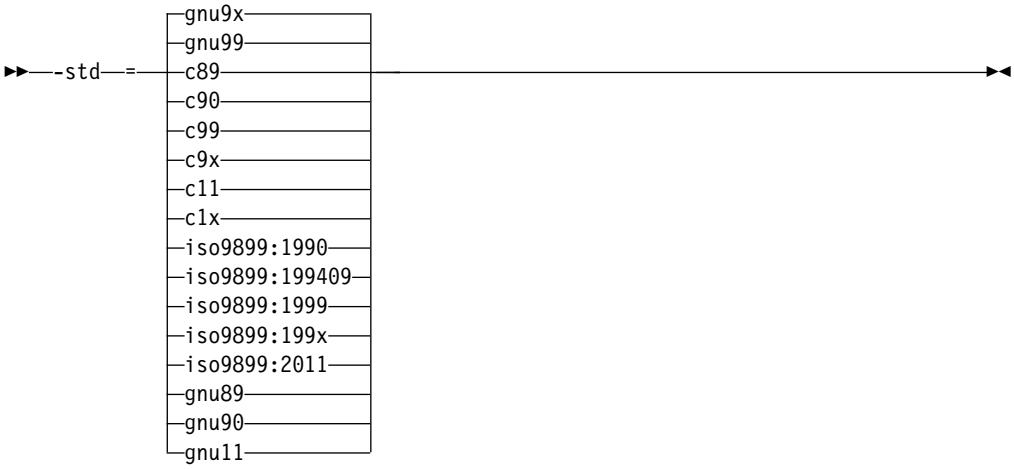
Determines whether source code and compiler options should be checked for conformance to a specific language standard, or subset or superset of a standard.

Syntax

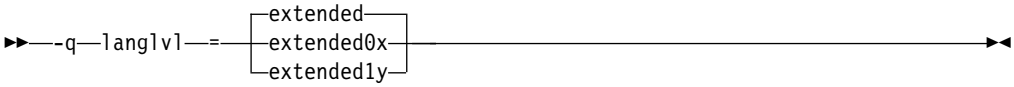
-qlanglvl syntax (C only)



-std syntax (C only)



-qlanglvl syntax (C++ only)



-std syntax (C++ only)

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 24. Mapping between the **-qlanglvl** *and* **-std** *suboptions (C only)*

-qlanglvl suboption	Mapping to -std suboption
stdc89	**c89 | c90 | iso9899:1990**
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 25. 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 269 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
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 68

-v, -V

Category

Listings, messages, and compiler information

Pragma equivalent

None.

Purpose

Reports the progress of compilation, by naming the programs being invoked and the options being specified to each program.

When the **-v** option is in effect, information is displayed in a comma-separated list. When the **-V** option is in effect, information is displayed in a space-separated list.

Syntax

Defaults

The compiler does not display the progress of the compilation.

Usage

The **-v** and **-V** options are overridden by the **-### (-#)** option.

Predefined macros

None.

Examples

To compile `myprogram.c` so you can watch the progress of the compilation and see messages that describe the progress of the compilation, the programs being invoked, and the options being specified, enter:

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

Related information

- "-### (-#) (pound sign)" on page 62

-w

Category

Listings, messages, and compiler information

Pragma equivalent

None.

Purpose

Suppresses warning messages.

Syntax

Defaults

All informational and warning messages are reported.

Usage

Informational and warning messages that supply additional information to a severe error are not disabled by this option.

Predefined macros

None.

Examples

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

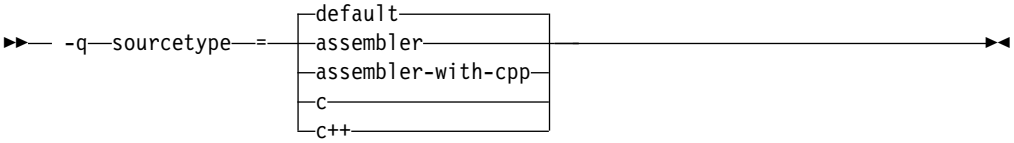
None.

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



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 63

-y

Category

Floating-point and integer control

Pragma equivalent

None.

Purpose

Specifies the rounding mode for the compiler to use when evaluating constant floating-point expressions at compile time.

Syntax

Defaults

- **-yn**

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, V13.1.3. 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
- -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
- -idirafter
- -imacros
- -include
- -iprefix
- -iquote
- -isysroot
- -isystem
- -iwithprefix
- -maltivec, -mno-altivec
- -mcpu
- -mtune
- -M
- -MD
- -MF
- -MG
- -MM
- -MMD

- -MP
- -MQ
- -MT
- -nodefaultlibs
- -nostartfiles
- -nostdinc
- -nostdinc++
- -Ofast
- -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
- -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

Chapter 5. Compiler pragmas reference

The following sections describe the available pragmas:

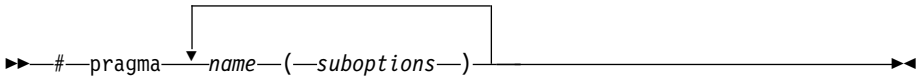
- "Pragma directive syntax"
- "Scope of pragma directives"
- "Supported GCC pragmas" on page 228
- "Supported IBM pragmas" on page 228

Pragma directive syntax

XL C/C++ supports the following forms of pragma directives:

#pragma *name*

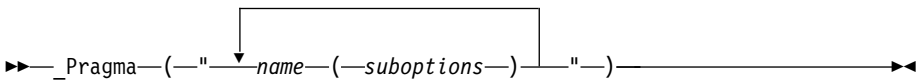
This form uses the following syntax:



The *name* is the pragma directive name, and the *suboptions* are any required or optional suboptions that can be specified for the pragma, where applicable.

_Pragma ("*name***")**

This form uses the following syntax:



For example, the statement:

```
_Pragma ( "pack(1)" )
```

is equivalent to:

```
#pragma pack(1)
```

For all forms of pragma statements, you can specify more than one *name* and *suboptions* in a single **#pragma** statement.

The *name* on a pragma is subject to macro substitutions, unless otherwise stated. The compiler ignores unrecognized pragmas, issuing an informational message indicating this.

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

© Copyright IBM Corp. 1996, 2015

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, V13.1.3. 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:

Category
The functional category to which the pragma belongs is listed here.

Purpose
This section provides a brief description of the effect of the pragma, and why you might want to use it.

Syntax
This section provides the syntax for the pragma. For convenience, the **#pragma** ***name*** form of the directive is used in each case. However, it is perfectly valid to use the alternate C99-style `_Pragma` operator syntax; see "Pragma directive syntax" on page 227 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.

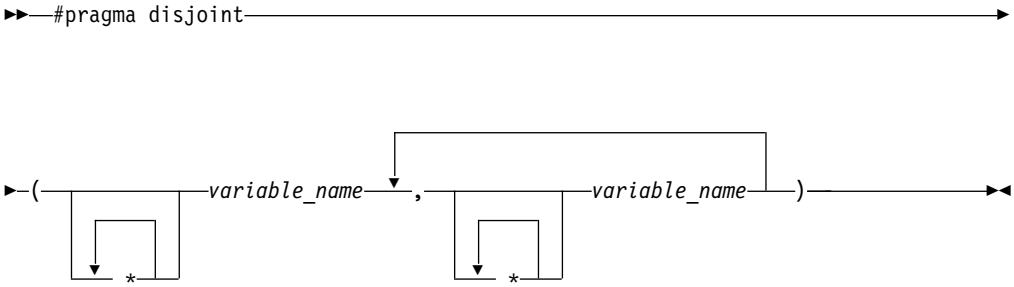
#pragma disjoint

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



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.

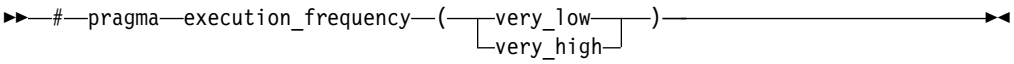
#pragma execution_frequency

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



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.

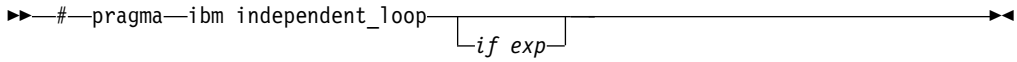
```
while (counter > 0) {
    #pragma execution_frequency(very_high)
    doSomething();
} /* This loop is very likely to be executed.    */

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



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

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 190

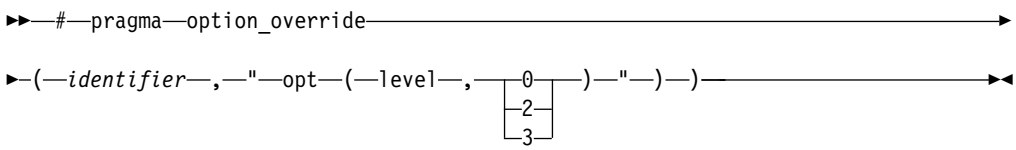
#pragma option_override

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



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 76
- "-qstrict" on page 199

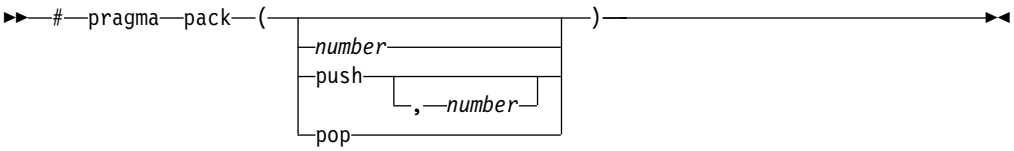
#pragma pack

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;
```

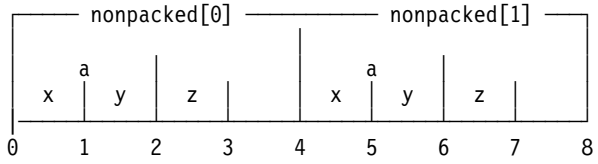
Default mapping:	**With #pragma pack(1):**
size of `s_t` = 16	size of `s_t` = 11
offset of `a` = 0	offset of `a` = 0

Default mapping:	**With #pragma pack(1):**
offset of b = 4	offset of b = 1
offset of c = 8	offset of c = 5
offset of d = 12	offset of d = 7
alignment of a = 1	alignment of a = 1
alignment of b = 4	alignment of b = 1
alignment of c = 2	alignment of c = 1
alignment of d = 4	alignment of d = 1

The following example defines a union uu containing a structure as one of its members, and declares an array of 2 unions of type uu:

```
union uu {
  short    a;
  struct {
    char x;
    char y;
    char z;
  } b;
};

union uu nonpacked[2];
```

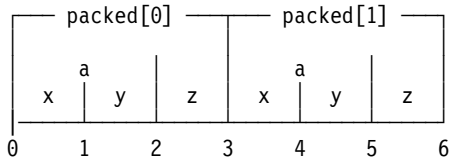
Since the largest alignment requirement among the union members is that of short a, namely, 2 bytes, one byte of padding is added at the end of each union in the array to enforce this requirement:



The next example uses **#pragma pack(1)** to set the alignment of unions of type uu to 1 byte:

```
#pragma pack(1)

union uu {
  short    a;
  struct {
    char x;
    char y;
    char z;
  } b;
};

union uu pack_array[2];
```

Now, each union in the array packed has a length of only 3 bytes, as opposed to the 4 bytes of the previous case:



Related information

- "-fpack-struct (-qalign)" on page 97
- "Using alignment modifiers" in the *XL C/C++ Optimization and Programming Guide*

#pragma reachable

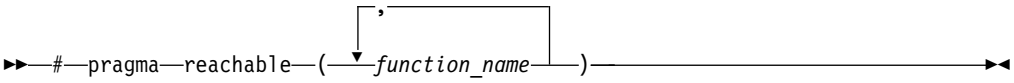
Purpose

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



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

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 145 in this document and "Using OpenMP directives" in the *IBM XL C/C++ Optimization and Programming Guide*.

Examples

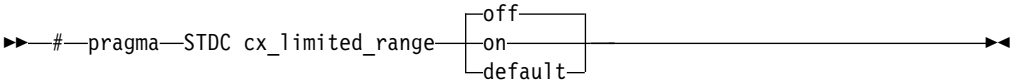
```
...
#pragma simd_level(10)
for (i=1; i<1000; i++) {
/* program code */

} ...
```

#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



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

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 108

Pragma directives for parallel processing

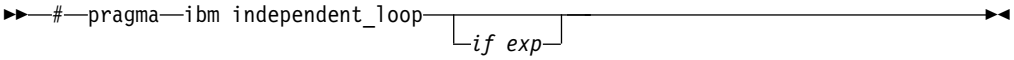
Parallel processing operations are controlled by pragma directives in your program source. The pragmas have effect only when parallelization is enabled with the **-qsmp** compiler option.

#pragma ibm independent_loop

Purpose

The **independent_loop** pragma explicitly states that the iterations of the chosen loop are independent, and that the iterations can be executed in parallel.

Syntax



where exp represents a scalar expression.

Usage

If the iterations of a loop are independent, you can put the pragma before the loop block. Then the compiler executes these iterations in parallel. When the exp argument is specified, the loop iterations are considered independent only if exp evaluates to TRUE at run time.

Notes:

- If the iterations of the chosen loop are dependent, the compiler executes the loop iterations sequentially no matter whether you specify the **independent_loop** pragma.
- To have an effect on a loop, you must put the **independent_loop** pragma immediately before this loop. Otherwise, the pragma is ignored.
- If several **independent_loop** pragmas are specified before a loop, only the last one takes effect.
- This pragma 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.

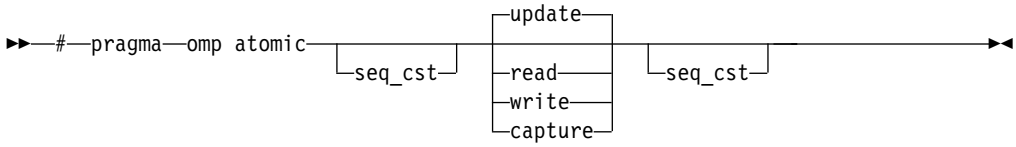
```
int a[1000], b[1000], c[1000];
int main(int k){
   if(k>0){
      #pragma ibm independent_loop if (k>2)
      for(int i=0; i<900; i++){
         a[i]=b[i]*c[i];
      }
   }
}
```

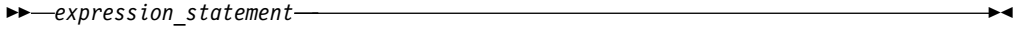
#pragma omp atomic
Purpose

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.

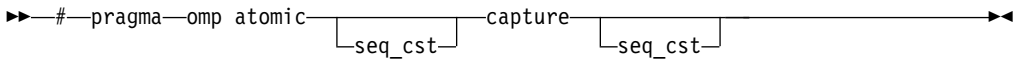
Syntax

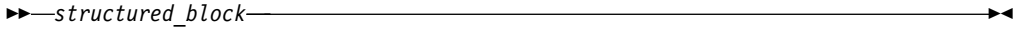
Syntax form 1





Syntax form 2





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.

Examples

Example 1: Atomic update

```
extern float x[], *p = x, y;

//Protect against race conditions among multiple updates.
#pragma omp atomic
x[index[i]] += y;

//Protect against race conditions with updates through x.
#pragma omp atomic
p[i] -= 1.0f;
```

Example 2: Atomic read, write, and update

```
extern int x[10];
extern int f(int);
int temp[10], i;

for(i = 0; i < 10; i++)
{
  #pragma omp atomic read
  temp[i] = x[f(i)];

  #pragma omp atomic write
  x[i] = temp[i]*2;
```

```
  #pragma omp atomic update
  x[i] *= 2;
}
```

Example 3: Atomic capture

```
extern int x[10];
extern int f(int);
int temp[10], i;

for(i = 0; i < 10; i++)
{
  #pragma omp atomic capture
  temp[i] = x[f(i)]++;

  #pragma omp atomic capture
  {
    temp[i] = x[f(i)]; //The two occurences of x[f(i)] must evaluate to the
    x[f(i)] -= 3; //same memory location, otherwise behavior is undefined.
  }
}
```

#pragma omp parallel

Purpose

The **omp parallel** directive explicitly instructs the compiler to parallelize the chosen block of code.

Syntax

Parameters

clause is any of the following clauses:

if (*exp***)**
When the `if` argument is specified, the program code executes in parallel only if the scalar expression represented by *exp* evaluates to a nonzero value at run time. Only one `if` clause can be specified.

private (*list***)**
Declares the scope of the data variables in *list* to be private to each thread. Data variables in *list* are separated by commas.

firstprivate (*list***)**
Declares the scope of the data variables in *list* to be private to each thread. Each new private object is initialized with the value of the original variable as if there was an implied declaration within the statement block. Data variables in *list* are separated by commas.

num_threads (*int_exp***)**
The value of *int_exp* is an integer expression that specifies the number of threads to use for the parallel region. If dynamic adjustment of the number of threads is also enabled, then *int_exp* specifies the maximum number of threads to be used.

shared (*list*)
Declares the scope of the comma-separated data variables in *list* to be shared across all threads.

default (shared | none)
Defines the default data scope of variables in each thread. Only one **default** clause can be specified on an **omp parallel** directive.

Specifying **default(shared)** is equivalent to stating each variable in a **shared(*list*)** clause.

Specifying **default(none)** requires that each data variable visible to the parallelized statement block must be explcitly listed in a data scope clause, with the exception of those variables that are:
- const-qualified,
- specified in an enclosed data scope attribute clause, or,
- used as a loop control variable referenced only by a corresponding **omp for** or **omp parallel for** directive.

copyin (*list*)
For each data variable specified in *list*, the value of the data variable in the master thread is copied to the thread-private copies at the beginning of the parallel region. Data variables in *list* are separated by commas.

Each data variable specified in the **copyin** clause must be a **threadprivate** variable.

reduction (*operator*: *list*)
Performs a reduction on all scalar variables in *list* using the specified *operator*. Reduction variables in *list* are separated by commas.

A private copy of each variable in *list* is created for each thread. At the end of the statement block, the final values of all private copies of the reduction variable are combined in a manner appropriate to the operator, and the result is placed back in the original value of the shared reduction variable. For example, when the `max` operator is specified, the original reduction variable value combines with the final values of the private copies by using the following expression:

```
original_reduction_variable = original_reduction_variable < private_copy ?
private_copy : original_reduction_variable;
```

For variables specified in the **reduction** clause, they must satisfy the following conditions:
- Must be of a type appropriate to the operator. If the `max` or `min` operator is specified, the variables must be one of the following types with or without `long`, `short`, `signed`, or `unsigned`:
  - C _Bool C
  - C++ bool C++
  - char
  - C++ wchar_t C++
  - int
  - float
  - double
- Must be shared in the enclosing context.
- Must not be const-qualified.
- Must not have pointer type.

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:

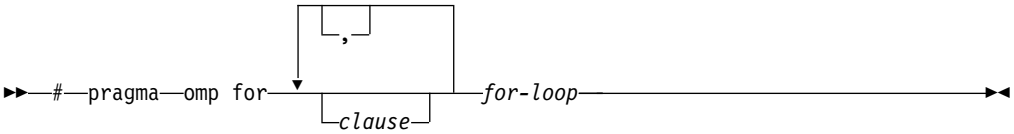
"OMP_NESTED" on page 27

"OMP_PROC_BIND" on page 31

#pragma omp for
Purpose

The **omp for** directive instructs the compiler to distribute loop iterations within the team of threads that encounters this work-sharing construct.

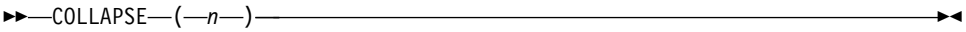
Syntax



Parameters

clause is any of the following clauses:

`collapse (`*`n`*`)`
Allows you to parallelize multiple loops in a nest without introducing nested parallelism.



- Only one collapse clause is allowed on a worksharing **for** or **parallel for** pragma.
- The specified number of loops must be present lexically. That is, none of the loops can be in a called subroutine.
- The loops must form a rectangular iteration space and the bounds and stride of each loop must be invariant over all the loops.
- If the loop indices are of different size, the index with the largest size will be used for the collapsed loop.
- The loops must be perfectly nested; that is, there is no intervening code nor any OpenMP pragma between the loops which are collapsed.

- The associated do-loops must be structured blocks. Their execution must not be terminated by an **break** statement.
- If multiple loops are associated to the loop construct, only an iteration of the innermost associated loop may be curtailed by a **continue** statement. If multiple loops are associated to the loop construct, there must be no branches to any of the loop termination statements except for the innermost associated loop.

Ordered construct
During execution of an iteration of a loop or a loop nest within a loop region, the executing thread must not execute more than one ordered region which binds to the same loop region. As a consequence, if multiple loops are associated to the loop construct by a collapse clause, the ordered construct has to be located inside all associated loops.

Lastprivate clause
When a lastprivate clause appears on the pragma that identifies a work-sharing construct, the value of each new list item from the sequentially last iteration of the associated loops, is assigned to the original list item even if a collapse clause is associated with the loop

Other SMP and performance pragmas
stream_unroll,**unroll**,**unrollandfuse**,**nounrollandfuse** pragmas cannot be used for any of the loops associated with the **collapse** clause loop nest.

private (*list***)**
Declares the scope of the data variables in *list* to be private to each thread. Data variables in *list* are separated by commas.

firstprivate (*list***)**
Declares the scope of the data variables in *list* to be private to each thread. Each new private object is initialized as if there was an implied declaration within the statement block. Data variables in *list* are separated by commas.

lastprivate (*list***)**
Declares the scope of the data variables in *list* to be private to each thread. The final value of each variable in *list*, if assigned, will be the value assigned to that variable in the last iteration. Variables not assigned a value will have an indeterminate value. Data variables in *list* are separated by commas.

reduction (*operator***:** *list***)**
Performs a reduction on all scalar variables in *list* using the specified *operator*. Reduction variables in *list* are separated by commas.

A private copy of each variable in *list* is created for each thread. At the end of the statement block, the final values of all private copies of the reduction variable are combined in a manner appropriate to the operator, and the result is placed back in the original value of the shared reduction variable. For example, when the `max` operator is specified, the original reduction variable value combines with the final values of the private copies by using the following expression:

```
original_reduction_variable = original_reduction_variable < private_copy ?
private_copy : original_reduction_variable;
```

For variables specified in the **reduction** clause, they must satisfy the following conditions:

- Must be of a type appropriate to the operator. If the `max` or `min` operator is specified, the variables must be one of the following types with or without `long`, `short`, `signed`, or `unsigned`:

  - C _Bool C
  - C++ bool C++
  - char
  - C++ wchar_t C++
  - int
  - float
  - double
- Must be shared in the enclosing context.
- Must not be const-qualified.
- Must not have pointer type.

ordered
Specify this clause if an ordered construct is present within the dynamic extent of the **omp for** directive.

schedule (*type***)**
Specifies how iterations of the **for** loop are divided among available threads. Acceptable values for *type* are:

auto With **auto**, scheduling is delegated to the compiler and runtime system. The compiler and runtime system can choose any possible mapping of iterations to threads (including all possible valid schedules) and these may be different in different loops.

dynamic
Iterations of a loop are divided into chunks of size **ceiling**(*number_of_iterations*/*number_of_threads*).

Chunks are dynamically assigned to active threads on a "first-come, first-do" basis until all work has been assigned.

dynamic,*n*
As above, except chunks are set to size *n*. *n* must be an integral assignment expression of value 1 or greater.

guided
Chunks are made progressively smaller until the default minimum chunk size is reached. The first chunk is of size **ceiling**(*number_of_iterations*/*number_of_threads*). Remaining chunks are of size **ceiling**(*number_of_iterations_left*/*number_of_threads*).

The minimum chunk size is 1.

Chunks are assigned to active threads on a "first-come, first-do" basis until all work has been assigned.

guided,*n*
As above, except the minimum chunk size is set to *n*; *n* must be an integral assignment expression of value 1 or greater.

runtime
Scheduling policy is determined at run time. Use the OMP_SCHEDULE environment variable to set the scheduling type and chunk size.

static Iterations of a loop are divided into chunks of size **ceiling**(*number_of_iterations*/*number_of_threads*). Each thread is assigned a separate chunk.

This scheduling policy is also known as *block scheduling*.

static,*n*

Iterations of a loop are divided into chunks of size *n*. Each chunk is assigned to a thread in *round-robin* fashion.

n must be an integral assignment expression of value 1 or greater.

This scheduling policy is also known as *block cyclic scheduling*.

Note: if *n*=1, iterations of a loop are divided into chunks of size 1 and each chunk is assigned to a thread in *round-robin* fashion. This scheduling policy is also known as *block cyclic scheduling*.

nowait

Use this clause to avoid the implied **barrier** at the end of the **for** directive. This is useful if you have multiple independent work-sharing sections or iterative loops within a given parallel region. Only one **nowait** clause can appear on a given **for** directive.

and where *for_loop* is a `for` loop construct with the following canonical shape:

```
for (init_expr; exit_cond; incr_expr)
 statement
```

where:

init_expr	takes the form:	`iv = b` `integer-type iv = b`

exit_cond	takes the form:	`iv <= ub`
		`iv <  ub`
		`iv >= ub`
		`iv >  ub`

incr_expr	takes the form:	`++iv`
		`iv++`
		`--iv`
		`iv--`
		`iv += incr`
		`iv -= incr`
		`iv = iv + incr`
		`iv = incr + iv`
		`iv = iv - incr`

and where:

iv	Iteration variable. The iteration variable must be a `signed integer` not modified anywhere within the `for` loop. It is implicitly made private for the duration of the `for` operation. If not specified as **lastprivate**, the iteration variable will have an indeterminate value after the operation completes.
b, ub, incr	Loop invariant signed integer expressions. No synchronization is performed when evaluating these expressions and evaluated side effects may result in indeterminate values.

Usage

This pragma must appear immediately before the loop or loop block directive to be affected.

Program sections using the **omp for** pragma must be able to produce a correct result regardless of which thread executes a particular iteration. Similarly, program correctness must not rely on using a particular scheduling algorithm.

The `for` loop iteration variable is implicitly made private in scope for the duration of loop execution. This variable must not be modified within the body of the `for` loop. The value of the increment variable is indeterminate unless the variable is specified as having a data scope of **lastprivate**.

An implicit barrier exists at the end of the `for` loop unless the **nowait** clause is specified.

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 ordered
Purpose

The **omp ordered** directive identifies a structured block of code that must be executed in sequential order.

Syntax

Usage

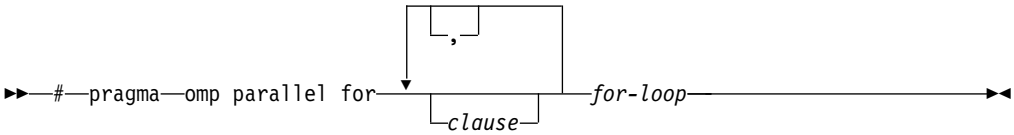
The **omp ordered** directive must be used as follows:

- It must appear within the extent of a **omp for** or **omp parallel for** construct containing an **ordered** clause.
- It applies to the statement block immediately following it. Statements in that block are executed in the same order in which iterations are executed in a sequential loop.
- An iteration of a loop must not execute the same **omp ordered** directive more than once.
- An iteration of a loop must not execute more than one distinct **omp ordered** directive.

#pragma omp parallel for
Purpose

The **omp parallel for** directive effectively combines the **omp parallel** and **omp for** directives. This directive lets you define a parallel region containing a single **for** directive in one step.

Syntax



Usage

With the exception of the **nowait** clause, clauses and restrictions described in the **omp parallel** and **omp for** directives also apply to the **omp parallel for** directive.

#pragma omp section, #pragma omp sections

Purpose

The **omp sections** directive distributes work among threads bound to a defined parallel region.

Syntax

Parameters

clause is any of the following clauses:

private (*list*)
Declares the scope of the data variables in *list* to be private to each thread. Data variables in *list* are separated by commas.

firstprivate (*list*)
Declares the scope of the data variables in *list* to be private to each thread. Each new private object is initialized as if there was an implied declaration within the statement block. Data variables in *list* are separated by commas.

lastprivate (*list*)
Declares the scope of the data variables in *list* to be private to each thread. The final value of each variable in *list*, if assigned, will be the value assigned to that variable in the last **section**. Variables not assigned a value will have an indeterminate value. Data variables in *list* are separated by commas.

reduction (*operator*: *list*)
Performs a reduction on all scalar variables in *list* using the specified *operator*. Reduction variables in *list* are separated by commas.

A private copy of each variable in *list* is created for each thread. At the end of the statement block, the final values of all private copies of the reduction variable are combined in a manner appropriate to the operator, and the result is placed back in the original value of the shared reduction variable. For example, when the `max` operator is specified, the original reduction variable value combines with the final values of the private copies by using the following expression:

```
original_reduction_variable = original_reduction_variable < private_copy ?
private_copy : original_reduction_variable;
```

For variables specified in the **reduction** clause, they must satisfy the following conditions:

- Must be of a type appropriate to the operator. If the `max` or `min` operator is specified, the variables must be one of the following types with or without `long`, `short`, `signed`, or `unsigned`:
  - C _Bool C
  - C++ bool C++
  - char
  - C++ wchar_t C++
  - int
  - float
  - double
- Must be shared in the enclosing context.
- Must not be const-qualified.
- Must not have pointer type.

nowait

Use this clause to avoid the implied **barrier** at the end of the **sections** directive. This is useful if you have multiple independent work-sharing sections within a given parallel region. Only one **nowait** clause can appear on a given **sections** directive.

Usage

The **omp section** directive is optional for the first program code segment inside the **omp sections** directive. Following segments must be preceded by an **omp section** directive. All **omp section** directives must appear within the lexical construct of the program source code segment associated with the **omp sections** directive.

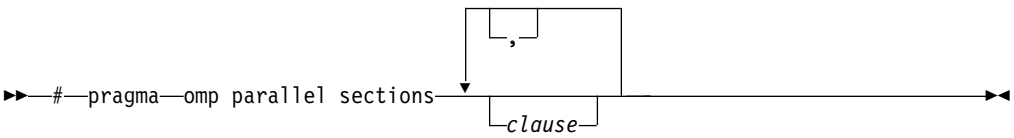
When program execution reaches a **omp sections** directive, program segments defined by the following **omp section** directive are distributed for parallel execution among available threads. A barrier is implicitly defined at the end of the larger program region associated with the **omp sections** directive unless the **nowait** clause is specified.

#pragma omp parallel sections

Purpose

The **omp parallel sections** directive effectively combines the **omp parallel** and **omp sections** directives. This directive lets you define a parallel region containing a single **sections** directive in one step.

Syntax



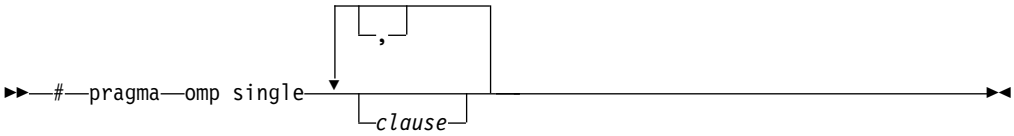
Usage

All clauses and restrictions described in the **omp parallel** and **omp sections** directives apply to the **omp parallel sections** directive.

#pragma omp single
Purpose

The **omp single** directive identifies a section of code that must be run by a single available thread.

Syntax



Parameters

clause is any of the following:

private (*list*)
Declares the scope of the data variables in *list* to be private to each thread. Data variables in *list* are separated by commas.

A variable in the **private** clause must not also appear in a **copyprivate** clause for the same **omp single** directive.

copyprivate (*list*)
Broadcasts the values of variables specified in *list* from one member of the team to other members. This occurs after the execution of the structured block associated with the **omp single** directive, and before any of the threads leave the barrier at the end of the construct. For all other threads in the team, each variable in the *list* becomes defined with the value of the corresponding variable in the thread that executed the structured block. Data variables in *list* are separated by commas. Usage restrictions for this clause are:

- A variable in the **copyprivate** clause must not also appear in a **private** or **firstprivate** clause for the same **omp single** directive.
- If an **omp single** directive with a **copyprivate** clause is encountered in the dynamic extent of a parallel region, all variables specified in the **copyprivate** clause must be private in the enclosing context.
- Variables specified in **copyprivate** clause within dynamic extent of a parallel region must be private in the enclosing context.
- A variable that is specified in the **copyprivate** clause must have an accessible and unambiguous copy assignment operator.
- The **copyprivate** clause must not be used together with the **nowait** clause.

firstprivate (*list*)
Declares the scope of the data variables in *list* to be private to each thread. Each new private object is initialized as if there was an implied declaration within the statement block. Data variables in *list* are separated by commas.

A variable in the **firstprivate** clause must not also appear in a **copyprivate** clause for the same **omp single** directive.

nowait
Use this clause to avoid the implied **barrier** at the end of the **single** directive. Only one **nowait** clause can appear on a given **single** directive. The **nowait** clause must not be used together with the **copyprivate** clause.

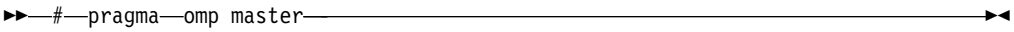
Usage

An implied barrier exists at the end of a parallelized statement block unless the **nowait** clause is specified.

#pragma omp master
Purpose

The **omp master** directive identifies a section of code that must be run only by the master thread.

Syntax



Usage

Threads other than the master thread will not execute the statement block associated with this construct.

No implied barrier exists on either entry to or exit from the master section.

#pragma omp critical
Purpose

The **omp critical** directive identifies a section of code that must be executed by a single thread at a time.

Syntax

where *name* can optionally be used to identify the critical region. Identifiers naming a critical region have external linkage and occupy a namespace distinct from that used by ordinary identifiers.

Usage

A thread waits at the start of a critical region identified by a given name until no other thread in the program is executing a critical region with that same name. Critical sections not specifically named by **omp critical** directive invocation are mapped to the same unspecified name.

#pragma omp barrier
Purpose

The **omp barrier** directive identifies a synchronization point at which threads in a parallel region will not execute beyond the **omp barrier** until all other threads in the team complete all explicit tasks in the region.

Syntax

Usage

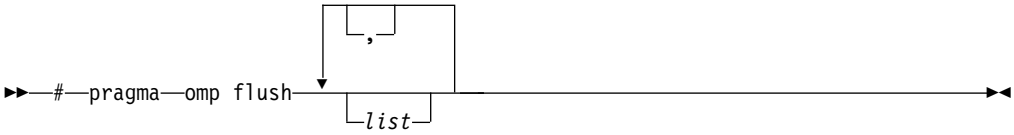
The **omp barrier** directive must appear within a block or compound statement. For example:

```
if (x!=0) {
   #pragma omp barrier    /* valid usage    */
}
if (x!=0)
   #pragma omp barrier    /* invalid usage  */
```

#pragma omp flush
Purpose

The **omp flush** directive identifies a point at which the compiler ensures that all threads in a parallel region have the same view of specified objects in memory.

Syntax



where *list* is a comma-separated list of variables that will be synchronized.

Usage

If *list* includes a pointer, the pointer is flushed, not the object being referred to by the pointer. If *list* is not specified, all shared objects are synchronized except those inaccessible with automatic storage duration.

An implied **flush** directive appears in conjunction with the following directives:

- **omp barrier**
- Entry to and exit from **omp critical**.
- Exit from **omp parallel**.
- Exit from **omp for**.
- Exit from **omp sections**.
- Exit from **omp single**.

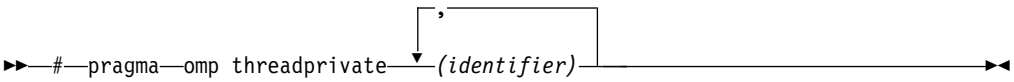
The **omp flush** directive must appear within a block or compound statement. For example:

```
if (x!=0) {
   #pragma omp flush    /* valid usage    */
}
if (x!=0)
   #pragma omp flush    /* invalid usage  */
```

#pragma omp threadprivate
Purpose

The **omp threadprivate** directive makes the named file-scope, namespace-scope, or static block-scope variables private to a thread.

Syntax



where *identifier* is a file-scope, name space-scope or static block-scope variable.

Usage

Each copy of an **omp threadprivate** data variable is initialized once prior to first use of that copy. If an object is changed before being used to initialize a **threadprivate** data variable, behavior is unspecified.

A thread must not reference another thread's copy of an **omp threadprivate** data variable. References will always be to the master thread's copy of the data variable when executing serial and master regions of the program.

Use of the **omp threadprivate** directive is governed by the following points:

- An **omp threadprivate** directive must appear at file scope outside of any definition or declaration.
- The **omp threadprivate** directive is applicable to static-block scope variables and may appear in lexical blocks to reference those block-scope variables. The directive must appear in the scope of the variable and not in a nested scope, and must precede all references to variables in its list.
- A data variable must be declared with file scope prior to inclusion in an **omp threadprivate** directive *list*.
- An **omp threadprivate** directive and its *list* must lexically precede any reference to a data variable found in that *list*.
- A data variable specified in an **omp threadprivate** directive in one translation unit must also be specified as such in all other translation units in which it is declared.
- Data variables specified in an **omp threadprivate** *list* must not appear in any clause other than the **copyin**, **copyprivate**, **if**, **num_threads**, and **schedule** clauses.
- The address of a data variable in an **omp threadprivate** *list* is not an address constant.
- A data variable specified in an **omp threadprivate** *list* must not have an incomplete or reference type.

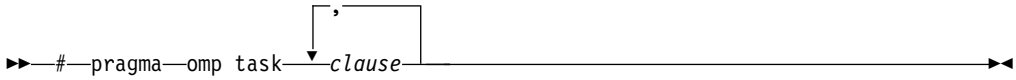
#pragma omp task
Purpose

The **task** pragma can be used to explicitly define a task.

Use the **task** pragma when you want to identify a block of code to be executed in parallel with the code outside the task region. The **task** pragma can be useful for

parallelizing irregular algorithms such as pointer chasing or recursive algorithms. The **task** directive takes effect only if you specify the **-qsmp** compiler option.

Syntax



Parameters

The *clause* parameter can be any of the following types of clauses:

default (shared | none)
Defines the default data scope of variable in each task. Only one `default` clause can be specified on an `omp task` directive.

Specifying `default(shared)` is equivalent to stating each variable in a `shared(list)` clause.

Specifying `default(none)` requires that each data variable visible to the construct must be explicitly listed in a data scope clause, with the exception of variables with the following attributes:

- Threadprivate
- Automatic and declared in a scope inside the construct
- Objects with dynamic storage duration
- Static data members
- The loop iteration variables in the associated for-loops for a work-sharing **for** or **parallel for** construct
- Static and declared in a scope inside the construct

final (*exp*)
If you specify a `final` clause and *exp* evaluates to a nonzero value, the generated task is a final task. All task constructs encountered inside a final task create final and included tasks.

You can specify only one `final` clause on the **task** pragma.

firstprivate (*list*)
Declares the scope of the data variables in *list* to be private to each thread. Each new private object is initialized with the value of the original variable as if there was an implied declaration within the statement block. Data variables in *list* are separated by commas.

if (*exp*)
When the `if` clause is specified, an undeferred task is generated if the scalar expression *exp* evaluates to a nonzero value. Only one `if` clause can be specified.

mergeable
If you specify a `mergeable` clause and the generated task is an undeferred task or included task, a merged task might be generated.

private (*list*)
Declares the scope of the data variables in *list* to be private to each thread. Data variables in *list* are separated by commas.

shared (*list*)
Declares the scope of the comma-separated data variables in *list* to be shared across all threads.

untied
When a task region is suspended, untied tasks can be resumed by any thread in a team. The `untied` clause on a task construct is ignored if either of the following conditions is a nonzero value:

- A `final` clause is specified on the same task construct and the `final` clause expression evaluates to a nonzero value.
- The task is an included task.

Usage

A final task is a task that makes all its child tasks become final and included tasks. A final task is generated when either of the following conditions is a nonzero value:

- A `final` clause is specified on a task construct and the `final` clause expression evaluates to nonzero value.
- The generated task is a child task of a final task.

An undeferred task is a task whose execution is not deferred with respect to its generating task region. In other words, the generating task region is suspended until the undeferred task has finished running. An undeferred task is generated when an `if` clause is specified on a task construct and the `if` clause expression evaluates to zero.

An included task is a task whose execution is sequentially included in the generating task region. In other words, an included task is undeferred and executed immediately by the encountering thread. An included task is generated when the generated task is a child task of a final task.

A merged task is a task that has the same data environment as that of its generating task region. A merged task might be generated when both the following conditions nonzero values:

- A `mergeable` clause is specified on a task construct.
- The generated task is an undeferred task or an included task.

The `if` clause expression and the `final` clause expression are evaluated outside of the task construct, and the evaluation order is not specified.

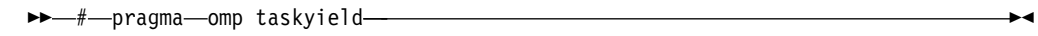
Related reference:

"#pragma omp taskwait" on page 261

#pragma omp taskyield
Purpose

The **omp taskyield** pragma instructs the compiler to suspend the current task in favor of running a different task. The **taskyield** region includes an explicit task scheduling point in the current task region.

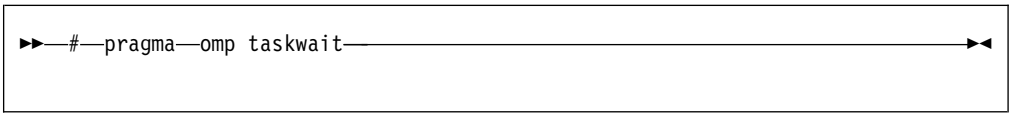
Syntax



#pragma omp taskwait
Purpose

Use the **taskwait** pragma to specify a *wait* for child tasks to be completed that are generated by the current task.

Syntax



Related reference:
"#pragma omp task" on page 258

Chapter 6. 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 266
- "Macros related to compiler features" on page 267

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 26. General predefined macros

Predefined macro name	**Description**	**Predefined value**
__BASE_FILE__	Indicates the name of the primary source file.	The fully qualified file name of the primary source file.
__DATE__	Indicates the date that the source file was preprocessed.	A character string containing the date when the source file was preprocessed.
__FILE__	Indicates the name of the preprocessed source file.	A character string containing the name of the preprocessed source file.
__FUNCTION__	Indicates the name of the function currently being compiled.	A character string containing the name of the function currently being compiled.
__LINE__	Indicates the current line number in the source file.	An integer constant containing the line number in the source file.
__SIZE_TYPE__	Indicates the underlying type of `size_t` on the current platform. Not protected.	`unsigned long`
__TIME__	Indicates the time that the source file was preprocessed.	A character string containing the time when the source file was preprocessed.

© Copyright IBM Corp. 1996, 2015

Table 26. General predefined macros (continued)

Predefined macro name	Description	Predefined value
__TIMESTAMP__	Indicates the date and time when the source file was last modified. The value changes as the compiler processes any include files that are part of your source program.	A character string literal in the form "*Day Mmm dd hh:mm:ss yyyy*", where: *Day* Represents the day of the week (`Mon`, `Tue`, `Wed`, `Thu`, `Fri`, `Sat`, or `Sun`). *Mmm* Represents the month in an abbreviated form (`Jan`, `Feb`, `Mar`, `Apr`, `May`, `Jun`, `Jul`, `Aug`, `Sep`, `Oct`, `Nov`, or `Dec`). *dd* Represents the day. If the day is less than 10, the first `d` is a blank character. *hh* Represents the hour. *mm* Represents the minutes. *ss* Represents the seconds. *yyyy* Represents the year.

Macros indicating the XL C/C++ compiler

Macros related to the XL C/C++ compiler are always predefined, and they are 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 27. 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
C++ __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
C++ __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

Table 27. Compiler-related predefined macros (continued)

Predefined macro name	Description	Predefined value
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.	3
__clang_minor__	Indicates the minor version number of the Clang compiler.	4
__clang_patchlevel__	Indicates the patch level number of the Clang compiler.	0
__clang_version__	Indicates the full version of the Clang compiler.	3.4 (tags/RELEASE_34/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 predefined by the compiler with the **-qxlcompatmacros** option. The option helps you migrate programs from IBM XL C/C++ for Linux V13.1 or earlier for big endian distributions to IBM XL C/C++ for Linux V13.1.3 for little endian distributions. However, it is recommended that you use the **-qnoxlcompatmacros** option to undefine these legacy macros when you migrate programs from V13.1.1 Linux for little endian distributions to V13.1.3 Linux for little endian distributions.

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 28. 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 269
- "Macros related to language levels" on page 269

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 29. 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	-maltivec (-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	-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

Table 29. 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__	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*
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	-maltivec (-qaltivec)
__VEC_ELEMENT_REG_ORDER__	Indicates the vector element order used in vector registers.	• __ORDER_LITTLE_ENDIAN__ when -qaltivec=le (-maltivec) is in effect • __ORDER_BIG_ENDIAN__ when -qaltivec=be is in effect	-maltivec (-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 enabling the feature is not in effect, then the macro is undefined.

Table 30. -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
_ARCH_PPCGR	Indicates that the application is targeted to run on Power processors with graphics support.	pwr8
_ARCH_PWR4	Indicates that the application is targeted to run on POWER4 or higher processors.	pwr8
_ARCH_PWR5	Indicates that the application is targeted to run on POWER5 or higher processors.	pwr8
_ARCH_PWR5X	Indicates that the application is targeted to run on POWER5+ or higher processors.	pwr8
_ARCH_PWR6	Indicates that the application is targeted to run on POWER6® or higher processors.	pwr8
_ARCH_PWR7	Indicates that the application is targeted to run on POWER7® , POWER7+™ or higher processors.	pwr8
_ARCH_PWR8	Indicates that the application is targeted to run on POWER8 processors.	pwr8

Related information

- "-mcpu (-qarch)" on page 123

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 31. 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.

Table 31. Predefined macros for language features (continued)

Predefined macro name	Description	Predefined when the following language level is in effect
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.)
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, V13.1.3. 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 32. Unsupported macros that are related to the platform

_BIG_ENDIAN, __BIG_ENDIAN__
_ILP32, __ILP32__
__THW_370__
__THW_BIG_ENDIAN__

Table 33. Unsupported macros related to compiler option settings

__LONGDOUBLE64
__IBM_GCC_ASM
__IBM_STDCPP_ASM
__TEMPINC__

Table 34. Unsupported macros related to architecture settings

_ARCH_PWR6E

Table 35. 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

Chapter 7. Compiler built-in functions

A built-in function is a coding extension to C and C++ that allows a programmer to use the syntax of C function calls and C variables to access the instruction set of the processor of the compiling machine. IBM Power architectures have special instructions that enable the development of highly optimized applications. Access to some Power instructions cannot be generated using the standard constructs of the C and C++ languages. Other instructions can be generated through standard constructs, but using built-in functions allows exact control of the generated code. Inline assembly language programming, which uses these instructions directly, is 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 274
- "Count zero functions" on page 275
- "Load functions" on page 276
- "Multiply functions" on page 277
- "Population count functions" on page 277
- "Rotate functions" on page 278
- "Store functions" on page 279
- "Trap functions" on page 280

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);
```

© Copyright IBM Corp. 1996, 2015

Assert functions

__assert1, __assert2
Purpose

Generates trap instructions.

Prototype

```
int __assert1 (int, int, int);

void __assert2 (int);
```

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

__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, __load4r
Purpose

Load Halfword Byte Reversed, Load Word Byte Reversed

Prototype

```
unsigned short __load2r (unsigned short*);

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);

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.

Rotate functions

__rdlam
Purpose

Rotate Double Left and AND with Mask

Rotates the contents of *rs* left *shift* bits, and ANDs the rotated data with the *mask*.

Prototype

unsigned long long __rdlam (unsigned long long *rs*, unsigned int *shift*, unsigned long long *mask*);

Parameters

mask
Must be a constant that represents a contiguous bit field.

__rldimi, __rlwimi
Purpose

Rotate Left Doubleword Immediate then Mask Insert, Rotate Left Word Immediate then Mask Insert

Rotates *rs* left *shift* bits then inserts *rs* into *is* under bit mask *mask*.

Prototype

unsigned long long __rldimi (unsigned long long *rs*, unsigned long long *is*, unsigned int *shift*, unsigned long long *mask*);

unsigned int __rlwimi (unsigned int *rs*, unsigned int *is*, unsigned int *shift*, unsigned int *mask*);

Parameters

shift
A constant value 0 to 63 (`__rldimi`) or 31 (`__rlwimi`).

mask
Must be a constant that represents a contiguous bit field.

__rlwnm
Purpose

Rotate Left Word then AND with Mask

Rotates *rs* left *shift* bits, then ANDs *rs* with bit mask *mask*.

Prototype

unsigned int __rlwnm (unsigned int *rs*, unsigned int *shift*, unsigned int *mask*);

Parameters

mask
Must be a constant that represents a contiguous bit field.

__rotatel4, __rotatel8
Purpose

Rotate Left Word, Rotate Left Doubleword

Rotates *rs* left *shift* bits.

Prototype

unsigned int __rotatel4 (unsigned int *rs*, unsigned int *shift*);

unsigned long long __rotatel8 (unsigned long long *rs*, unsigned long long *shift*);

Store functions

__store2r, __store4r
Purpose

Store 2/4-byte Reversal

Prototype

void __store2r (unsigned short, unsigned short*);

void __store4r (unsigned int, unsigned int*);

__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

TO A value of 0 to 31 inclusive. Each bit position, if set, indicates one or more of the following possible conditions:

0 (high-order bit)
a is less than *b*, using signed comparison.

1 *a* is greater than *b*, using signed comparison.

2 *a* is equal to *b*

3 *a* is less than *b*, using unsigned comparison.

4 (low-order bit)
a is greater than *b*, using unsigned comparison.

__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 273
- "Conversion functions"
- "FPSCR functions" on page 284
- "Multiply-add/subtract functions" on page 286
- "Reciprocal estimate functions" on page 287
- "Rounding functions" on page 287
- "Select functions" on page 289
- "Square root functions" on page 289
- "Software division functions" on page 289

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);
```

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.

FPSCR functions

__mtfsb0
Purpose

Move to Floating-Point Status/Control Register (FPSCR) Bit 0

Sets bit *bt* of the FPSCR to 0.

Prototype

void __mtfsb0 (unsigned int *bt*);

Parameters

bt Must be a constant with a value of 0 to 31.

__mtfsb1
Purpose

Move to FPSCR Bit 1

Sets bit *bt* of the FPSCR to 1.

Prototype

void __mtfsb1 (unsigned int *bt*);

Parameters

bt Must be a constant with a value of 0 to 31.

__mtfsf
Purpose

Move to FPSCR Fields

Places the contents of *frb* into the FPSCR under control of the field mask specified by *flm*. The field mask *flm* identifies the 4bit fields of the FPSCR affected.

Prototype

void __mtfsf (unsigned int *flm*, unsigned int *frb*);

Parameters

flm
Must be a constant 8-bit mask.

__mtfsfi
Purpose

Move to FPSCR Field Immediate

Places the value of *u* into the FPSCR field specified by *bf*.

Prototype

void __mtfsfi (unsigned int *bf*, unsigned int *u*);

Parameters

bf Must be a constant with a value of 0 to 7.

u Must be a constant with a value of 0 to 15.

__readflm
Purpose

Returns a 64-bit double precision floating point, whose 32 low order bits contain the contents of the FPSCR. The 32 low order bits are bits 32 - 63 counting from the highest order bit.

Prototype

double __readflm (void);

__setflm
Purpose

Takes a double precision floating-point number and places the lower 32 bits in the FPSCR. The 32 low order bits are bits 32 - 63 counting from the highest order bit.

Returns the previous contents of the FPSCR.

Prototype

```
double __setflm (double);
```

__setrnd
Purpose

Sets the rounding mode.

Prototype

```
double __setrnd (int mode);
```

Parameters

The allowable values for *mode* are:
- 0 — round to nearest
- 1 — round to zero
- 2 — round to +infinity
- 3 — round to -infinity

Multiply-add/subtract functions

__fmadd, __fmadds
Purpose

Floating Multiply-Add, Floating Multiply-Add Single

Multiplies the first two arguments, adds the third argument, and returns the result.

Prototype

```
double __fmadd (double, double, double);

float __fmadds (float, float, float);
```

__fmsub, __fmsubs
Purpose

Floating Multiply-Subtract, Floating Multiply-Subtract Single

Multiplies the first two arguments, subtracts the third argument and returns the result.

Prototype

```
double __fmsub (double, double, double);

float __fmsubs (float, float, float);
```

__fnmadd, __fnmadds
Purpose

Floating Negative Multiply-Add, Floating Negative Multiply-Add Single

Multiplies the first two arguments, adds the third argument, and negates the result.

Prototype

```
double __fnmadd (double, double, double);

float __fnmadds (float, float, float);
```

__fnmsub, __fnmsubs
Purpose

Floating Negative Multiply-Subtract

Multiplies the first two arguments, subtracts the third argument, and negates the result.

Prototype

```
double __fnmsub (double, double, double);

float __fnmsubs (float, float, float);
```

Reciprocal estimate functions

See also "Square root functions" on page 289.

__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

__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);
```

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.

BCD add and subtract

__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 nonnegative value and *ps* is 0, the sign is set to 0b1100 (0xC).
- If the result is a nonnegative 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).

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 nonnegative value and *ps* is 0, the sign is set to 0b1100 (0xC).
- If the result is a nonnegative 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).

Prototype

vector unsigned char __bcdsub (vector unsigned char *a*, vector unsigned char *b*, long *ps*);

Parameters

ps A compile-time known constant.

BCD test add and subtract for overflow

__bcdadd_ofl
Purpose

Returns 1 if the corresponding BCD add operation results in an overflow, or 0 otherwise.

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.

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.

Prototype

```
long __bcd_invalid (vector unsigned char a);
```

BCD comparison

__bcdcmpeq
Purpose

Returns 1 if the BCD value *a* is equal to *b*, or 0 otherwise.

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.

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.

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.

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.

Prototype

long __bcdcmplt (vector unsigned char *a*, vector unsigned char *b*);

BCD load and store

__vec_ldrmb
Purpose

Loads a string of bytes into vector register, right-justified. Sets the leftmost elements (16-*cnt*) to 0.

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.

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.

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 295
- "Compare and swap functions" on page 296
- "Fetch functions" on page 297
- "Load functions" on page 298
- "Store functions" on page 299
- "Synchronization functions" on page 300

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 442 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 303

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 122

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.

Related reference:

"-maltivec (-qaltivec)" on page 122

vec_abs

Purpose

Returns a vector containing the absolute values of the contents of the given vector.

Syntax

```
d=vec_abs(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 36. 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_abss

Purpose

Returns a vector containing the saturated absolute values of the elements of a given vector.

Syntax

```
d=vec_abss(a)
```

Result and argument types

The following table describes the types of the returned value and the function argument.

Table 37. 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.

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 38. 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.

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.

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 39. 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
vector signed int	vector bool int	vector signed int
	vector signed int	vector bool int
		vector signed int

Table 39. Types of the returned value and function arguments (continued)

d	a	b
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.

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.

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.

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.

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.

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 40. 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.

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 41. 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.

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 42. 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.

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 43. Types of the returned value and the function arguments

d	a	b
int	vector float	vector float

Result value

The result is 1 if all 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 0.

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.

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 44. 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.

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 45. 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`.

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 46. 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.

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 47. 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.

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 48. 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.

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 49. 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.

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 50. 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.

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 51. 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`).

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 52. 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.

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 53. 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
vector signed int	vector bool int	vector signed int
	vector signed int	vector signed int
		vector bool int

Table 53. 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 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.

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 54. 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

Table 54. Result and argument types (continued)

d	a	b
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.

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 55. 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.

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 56. 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.

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 57. 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.

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 58. 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.

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 59. 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`.

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 60. 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.

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 61. 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.

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 62. 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.

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 63. 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.

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 64. 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.

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 65. 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`).

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 66. 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.

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 67. Types of the returned value and the function arguments

d	a	b
int	vector float	vector float

Result value

The result is 1 if 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`. Otherwise, the result is 0.

vec_avg

Purpose

Returns a vector containing the rounded average of each set of corresponding elements of two given vectors.

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 68. 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 in the specified order, and places them in the specified order in the rightmost 16 bits of the leftmost doubleword of the result vector register, with the rest of the result zeroed.

Syntax

```
d=vec_bperm(a, b)
```

Result and argument types

The type of d, a, and b must be vector unsigned char.

Result value

For each i (0 <= i < 16), let index denote the byte value of the ith element of b.

If index is greater than or equal to 128, bit 48+i of the result is set to 0.

If index is smaller than 128, bit 48+i of the result is set to the value of the indexth bit of input a.

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 392.

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`.

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`.

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.

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 69. 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 the value of the corresponding element of `a` is 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 value of the corresponding element of the result is 0 if the absolute value of the corresponding element of `a` is equal to the value of the corresponding element of `b`, negative if it is greater than the value of the corresponding element of `b`, and positive if it is less than the value of the corresponding element of `b`.
- 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.

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 70. 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.

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 71. 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.

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 72. 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

Table 72. Result and argument types (continued)

d	a	b
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.

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 73. 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 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.

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 74. 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_cntlz

Purpose

Computes the count of leading zero bits of each element of the input.

Syntax

```
d=vec_cntlz(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 75. Result and argument types

d	a
vector unsigned char	vector unsigned char
	vector signed char
vector unsigned short	vector unsigned short
	vector signed short
vector unsigned int	vector unsigned int
	vector signed int
vector unsigned long long	vector unsigned 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`.

vec_cpsgn

Purpose

Returns a vector by copying the sign of the elements in vector `a` to the sign of the corresponding elements in vector `b`.

Syntax

```
d=vec_cpsgn(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 76. Result and argument types

d	a	b
vector float	vector float	vector float
vector double	vector double	vector double

vec_ctd

Purpose

Converts the type of each element in `a` from integer to floating-point single precision and divides the result by 2 to the power of `b`.

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 77. 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.

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 78. 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 the argument `a` is a signed long long or unsigned long long vector.

vec_cts

Purpose

Converts a vector of floating-point numbers into a vector of signed fixed-point numbers.

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 79. 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.

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 80. 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.

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 81. 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.

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 82. 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.

Syntax

```
d=vec_cvf(a)
```

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
vector float	vector double

Table 83. Result and argument types (continued)

d	a
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.

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 84. 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`.

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.

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 using this built-in function, the specified data stream is relatively persistent.

Syntax

```
vec_dst(a, b, c)
```

Result and argument types

This function does not return any value. The following table describes the types of the function arguments.

Table 85. 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.

Syntax

```
vec_dstst(a, b, c)
```

Result and argument types

This function does not return any value. The following table describes the types of the function arguments.

Table 86. 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.

Syntax

```
vec_dststt(a, b, c)
```

Result and argument types

This function does not return a value. The following table describes the types of the function arguments.

Table 87. 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.

Syntax

```
vec_dstt(a, b, c)
```

Result and argument types

This function does not return a value. The following table describes the types of the function arguments.

Table 88. 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.

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 89. 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

Table 89. Types of the returned value and function arguments (continued)

d	a	b
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.

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_extract

Purpose

Returns the value of element `b` from the vector `a`.

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 90. 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 the vector to determine the element position.

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 392.

vec_gbb

Purpose

Performs a gather-bits-by-bytes operation on the input.

Syntax

```
d=vec_gbb(a)
```

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**
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 `j`th bit of the `i`th byte element of the result is set to the value of the `i`th bit of the `j`th 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`.

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 92. 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_ld

Purpose

Loads a vector from the given memory address.

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 93. 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 93. 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

a is added to the address of 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.

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 94. 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.

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 95. 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_loge

Purpose

Returns a vector containing estimates of the base-2 logarithms of the corresponding elements of the given vector.

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.

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 96. 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 `a` and the address of `b`, modulo 16. Each successive element contains the previous element's value plus 1.

vec_lvsr

Purpose

Returns a vector useful for aligning non-aligned data.

Syntax

```
d=vec_lvsr(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 97. 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 `a` and the address of `b`, modulo 16. The first element of the result vector contains the value 16 minus the effective address. Each successive element contains the previous element's value 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.

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 98. 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.

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.

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 99. 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

Table 99. 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 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 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.

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 100. 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 366

vec_mergeh

Purpose

Merges the most significant halves of two vectors.

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 101. 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 122

"vec_mergel"

Related information:

Vector element order toggling

vec_mergel

Purpose

Merges the least significant halves of two vectors.

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 102. 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 122

"vec_mergeh" on page 365

Related information:

Vector element order toggling

vec_mergeo

Purpose

Merges the values of odd-numbered elements of two vectors.

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 103. 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 364

vec_mfvscr

Purpose

Copies the contents of the Vector Status and Control Register into the result vector.

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.

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 104. 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.

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 105. 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.

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.

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 106. 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.

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 107. 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.

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 108. 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 2`n` to 2`n`+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.

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.

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 109. 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.

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 110. 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 2`n` of `a` and the value of element 2`n` 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.

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 111. 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 2`n`+1 of `a` and the value of element 2`n`+1 of `b`.

vec_nabs

Purpose

Returns a vector containing the results of performing a negative-absolute operation using the given vector.

Syntax

```
d=vec_nabs(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 112. 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.

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 113. 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

Table 113. Types of the returned value and function arguments (continued)

d	a	b
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 `i`th 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`.

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`.

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.

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 largest absolute value is selected.

Related reference:

"vec_round" on page 391

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.

Syntax

```
d=vec_neg(a)
```

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**
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.

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 115. 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.

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 116. 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

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.

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 117. 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 117. 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 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 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.

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 118. 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
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.

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 119. 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

Table 119. Types of the returned value and function arguments (continued)

d	a	b
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.

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 120. 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 122

Related information:

Vector element order toggling

vec_packpx

Purpose

Packs information from each element of two vectors into the result vector.

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 121. 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.

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 122. 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.

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 123. 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_perm

Purpose

Returns a vector that contains some elements of two vectors, in the order specified by a third vector.

Syntax

```
d=vec_perm(a, b, c)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 124. 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_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.

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 125. 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.

Syntax

```
d=vec_popcnt(a)
```

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**
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`.

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 127. 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.

Syntax

```
d=vec_re(a)
```

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
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.

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 387

"vec_div" on page 350

vec_revb

Purpose

Returns a vector that contains the bytes of the corresponding element of the argument in the reverse byte order.

Syntax

```
d=vec_revb(a)
```

Result and argument types

The following table describes the types of the returned value and the function argument.

Table 129. 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.

Syntax

```
d=vec_reve(a)
```

Result and argument types

The following table describes the types of the returned value and the function argument.

Table 130. 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.

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 391

vec_rl

Purpose

Rotates each element of a vector left by a given number of bits.

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 131. 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_round

Purpose

Returns a vector containing the rounded values of the corresponding elements of the given vector.

Syntax

```
d=vec_round(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 132. 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.

Syntax

```
d=vec_roundc(a)
```

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**
vector float	vector float
vector double	vector double

Related information

"vec_rint" on page 390

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`.

Syntax

```
d=vec_roundm(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 134. Result and argument types

d	**a**
vector float	vector float
vector double	vector double

Related reference:

"vec_floor" on page 355

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`.

Syntax

```
d=vec_roundp(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 135. Result and argument types

d	**a**
vector float	vector float
vector double	vector double

Related reference:

"vec_ceil" on page 339

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`.

Syntax

```
d=vec_roundz(a)
```

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**
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 416

vec_rsqrt

Purpose

Returns a vector that contains estimates of the reciprocal square roots of the corresponding elements of the given vector.

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 the reciprocal square root of the corresponding element of `a` by using the vector reciprocal square root estimate instruction and iterative refinement.

Related reference:

"vec_rsqrte"

vec_rsqrte

Purpose

Returns a vector containing estimates of the reciprocal square roots of the corresponding elements of the given vector.

Syntax

```
d=vec_rsqrte(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 137. 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`.

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`.

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 138. 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.

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 139. 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 (∑). 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 (σ0 or ∑0) or sigma-1 (σ1 or ∑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:
  - σ0(x[*i*]), if `b` is 0 and bit *i* of the 4-bit `c` is 0
  - σ1(x[*i*]), if `b` is 0 and bit *i* of the 4-bit `c` is 1
  - ∑0(x[*i*]), if `b` is nonzero and bit *i* of the 4-bit `c` is 0
  - ∑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:
  - σ0(x[*i*]), if `b` is 0 and bit 2**i* of the 4-bit `c` is 0
  - σ1(x[*i*]), if `b` is 0 and bit 2**i* of the 4-bit `c` is 1
  - ∑0(x[*i*]), if `b` is nonzero and bit 2**i* of the 4-bit `c` is 0
  - ∑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.

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 140. 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.

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 141. 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

Returns a vector by concatenating `a` and `b`, and then left-shifting the result vector by multiples of 4 bytes. `c` specifies the offset for the shifting operation.

Syntax

```
d=vec_sldw(a, b, c)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 142. 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

After left-shifting the concatenated `a` and `b` by multiples of 4 bytes specified by `c`, the function takes the four leftmost 4-byte values and forms the result vector.

vec_sll

Purpose

Left shifts a vector by a given number of bits.

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 143. 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.

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 144. 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_splat

Purpose

Returns a vector that has all of its elements set to a given value.

Syntax

```
d=vec_splat(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 145. 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

The value of each element of the result is the value of the element of `a` specified by `b`.

vec_splats

Purpose

Returns a vector of which the value of each element is set to `a`.

Syntax

```
d=vec_splats(a)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 146. 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
vector unsigned long long	unsigned long long
vector float	float
vector double	double

vec_splat_s8

Purpose

Returns a vector with all elements equal to the given value.

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 147. 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 with all elements equal to the given value.

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 148. 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 with all elements equal to the given value.

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 149. 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 with all elements equal to the given value.

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 150. 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 with all elements equal to the given value.

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 151. 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 with all elements equal to the given value.

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 152. Types of the returned value and function arguments

d	**a**[1]
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.

Syntax

```
d=vec_sqrt(a)
```

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**
vector float	vector float
vector double	vector double

vec_sr

Purpose

Performs a right shift for each element of a vector.

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 154. 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.

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 155. 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	

Table 155. Result and argument types (continued)

d	a	b
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.

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 156. 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 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.

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 157. 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 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_st

Purpose

Stores a vector to memory at the given address.

Syntax

```
vec_st(a, b, c)
```

Result and argument types

The vec_st function returns nothing. b is added to the address of 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 158. Data type of function returned value and arguments

a	b	c
vector unsigned int	int	unsigned long*
vector signed int		signed long*
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.

Syntax

```
vec_ste(a,b,c)
```

Result and argument types

This function does not return a value. The following table describes the types of the function arguments.

Table 159. 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 `b` 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.

Syntax

```
vec_stl(a,b,c)
```

Result and argument types

This function does not return a value. The following table describes the types of the function arguments.

Table 160. 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.

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 161. 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.

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.

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.

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.

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.

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.

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 162. 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
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.

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.

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 163. Types of the returned value and function arguments

d	a	b
vector signed int	vector signed char	vector signed int
vector signed int	vector signed short	vector signed int
vector unsigned int	vector unsigned char	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`, 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.

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_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 393.

vec_unpackh

Purpose

Unpacks the most significant half of a vector into a vector with larger elements.

Syntax

```
d=vec_unpackh(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 signed short	vector signed char
vector signed int	vector signed short
vector signed long long	vector signed int

Table 164. Result and argument types (continued)

d	a
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 122

Related information:

Vector element order toggling

vec_unpackl

Purpose

Unpacks the least significant half of a vector into a vector with larger elements.

Syntax

```
d=vec_unpackl(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 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 122

Related information:

Vector element order toggling

vec_vclz

Purpose

Computes the count of leading zero bits of each element of the given vector.

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 345

vec_vgbbd

Purpose

Performs a gather-bits-by-bytes operation on the given vector.

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 `j`th bit of the `i`th byte element of the result is set to the value of the `i`th bit of the `j`th byte element of the input.

Related reference:
"vec_gbb" on page 356

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.

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 166. 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 `-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.

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 167. 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_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.

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 168. 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 122

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.

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 169. 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.

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 170. 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 122

Related information:

Vector element order toggling

vec_xor

Purpose

Performs a bitwise XOR of the given vectors.

Syntax

```
d=vec_xor(a, b)
```

Result and argument types

The following table describes the types of the returned value and the function arguments.

Table 171. 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

Table 171. Result and argument types (continued)

d	a	b
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 XOR of `a` and `b`.

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.

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 172. 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.

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 173. 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.

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 174. 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 122

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.

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 175. 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 122

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 176. 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 177. 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, V13.1.3 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 178 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 179 on page 440.

Table 178. 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 178. 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 179. 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, V13.1.3 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 **-O** 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 **-O** 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, V13.1.3 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 **-O2** or higher optimization level, the compiler issues a warning message if it can determine at compile time that the object will always be overflown; 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 overflown.

The supported built-in functions for common string operation functions are described in the following table.

Table 180. 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 181. 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 181. 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 443
- "Memory-related functions" on page 445

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 supervisor mode only.

__mfspr
Purpose

Move from Special-Purpose Register

Returns the value of given special purpose register.

Prototype

```
unsigned long __mfspr (const int registerNumber);
```

Parameters

registerNumber
The number of the special purpose register whose value is to be returned. The *registerNumber* must be known at compile time.

__mtmsr
Purpose

Move to Machine State Register

Moves the contents of bits 32 to 62 of the designated GPR into the MSR.

Prototype

```
void __mtmsr (unsigned long value);
```

Parameters

value
The bitwise OR result of bits 48 and 49 of *value* is placed into MSR_{48}. The bitwise OR result of bits 58 and 49 of *value* is placed into MSR_{58}. The bitwise OR result of bits 59 and 49 of *value* is placed into MSR_{59}. Bits 32:47, 49:50, 52:57, and 60:62 of *value* are placed into the corresponding bits of the MSR.

Usage

Execution of this instruction is privileged and restricted to supervisor mode only.

__mtspr
Purpose

Move to Special-Purpose Register

Sets the value of a special purpose register.

Prototype

void __mtspr (const int *registerNumber*, unsigned long *value*);

Parameters

registerNumber
The number of the special purpose register whose value is to be set. The *registerNumber* must be known at compile time.

value
Must be known at compile time.

Memory-related functions

__alloca
Purpose

Allocates space for an object. The allocated space is put on the stack and freed when the calling function returns.

Prototype

void* __alloca (size_t *size*)

Parameters

size
An integer representing the amount of space to be allocated, measured in bytes.

__builtin_frame_address, __builtin_return_address
Purpose

Returns the address of the stack frame, or return address, of the current function, or of one of its callers.

Prototype

void* __builtin_frame_address (unsigned int *level*);

void* __builtin_return_address (unsigned int *level*);

Parameters

level
A constant literal indicating the number of frames to scan up the call stack. The *level* must range from 0 to 63. A value of 0 returns the frame or return address of the current function, a value of 1 returns the frame or return address of the caller of the current function and so on.

Return value

Returns 0 when the top of the stack is reached. Optimizations such as inlining may affect the expected return value by introducing extra stack frames or fewer stack frames than expected. If a function is inlined, the frame or return address corresponds to that of the function that is returned to.

__mem_delay
Purpose

The **__mem_delay** built-in function specifies how many delay cycles there are for specific loads. These specific loads are delinquent loads with a long memory access latency because of cache misses.

When you specify which load is delinquent the compiler takes that information and carries out optimizations such as data prefetching. 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 177.

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 177

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.

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" on page 449
- "Transaction inquiry functions" on page 450

__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 448
- "Transaction inquiry functions" on page 450

Transaction abort functions

__TM_abort
Purpose

Aborts a transaction with failure code 0.

Prototype

```
void __TM_abort ();
```

Related information

- "__TM_named_abort" on page 450

__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 449

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 452

__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 ();

Chapter 8. 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_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_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.

© Copyright IBM Corp. 1996, 2015

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 31

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 457

Related information:

"OMP_SCHEDULE" on page 35

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:

"omp_get_schedule" on page 456

Related information:

"OMP_SCHEDULE" on page 35

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_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 nonnegative integer.

Prototype

```
int omp_get_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_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_active_level

Purpose

Returns the number of nested, active parallel regions enclosing the task that contains the call. The routine always returns a nonnegative integer, and returns 0 if it is called from the sequential part of the program.

Prototype

```
int omp_get_active_level(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_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_threads

Purpose

Returns the number of threads currently in the team executing the parallel region from which it is called.

Prototype

```
int omp_get_num_threads (void);
```

omp_set_num_threads

Purpose

Overrides the setting of the *OMP_NUM_THREADS* environment variable, and specifies the number of threads to use for a subsequent parallel region by setting the first value of *num_list* for *OMP_NUM_THREADS*.

Prototype

```
void omp_set_num_threads (int num_threads);
```

Parameters

num_threads
Must be a positive integer.

Usage

If the `num_threads` clause is present, then for the parallel region it is applied to, it supersedes the number of threads requested by this function or the

OMP_NUM_THREADS environment variable. Subsequent parallel regions are not affected by it.

omp_get_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_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_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_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_in_final

Purpose

Returns a nonzero integer value if the function is called in a final task region; otherwise, it returns 0.

Prototype

int omp_in_final(void);

omp_in_parallel

Purpose

Returns non-zero if it is called within the dynamic extent of a parallel region executing in parallel; otherwise, returns 0.

Prototype

int omp_in_parallel (void);

omp_set_dynamic

Purpose

Enables or disables dynamic adjustment of the number of threads available for execution of parallel regions.

Prototype

void omp_set_dynamic (int *dynamic_threads*);

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_get_dynamic

Purpose

Returns non-zero if dynamic thread adjustment is enabled and returns 0 otherwise.

Prototype

int omp_get_dynamic (void);

omp_set_nested

Purpose

Enables or disables nested parallelism.

Prototype

void omp_set_nested (int *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_get_nested

Purpose

Returns non-zero if nested parallelism is enabled and 0 if it is disabled.

Prototype

int omp_get_nested (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_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_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_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**.

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_get_wtime

Purpose

Returns the time elapsed from a fixed starting time.

Prototype

```
double omp_get_wtime (void);
```

Usage

The value of the fixed starting time is determined at the start of the current program, and remains constant throughout program execution.

omp_get_wtick

Purpose

Returns the number of seconds between clock ticks.

Prototype

```
double omp_get_wtick (void);
```

Usage

The value of the fixed starting time is determined at the start of the current program, and remains constant throughout program execution.

Notices

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

© Copyright IBM Corp. 1996, 2015

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, 2015.

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.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Index

Special characters

--help compiler option 63
--version (-qversion) compiler option 64
-fstandalone-debug compiler option 99
-ftrapping-math (-qflttrap) compiler option 103
-qhelp compiler option 63
-qlistfmt compiler option 163
-qreport compiler option 180
-qsaveopt compiler option 187
-qsmp compiler option 193
-qxlcompatmacros 206
*_chk 440

A

alias 99
  -qalias compiler option 99
  pragma disjoint 229
alignment 97
  -fpack-struct (-qalign) compiler option 97
  pragma align 97
  pragma pack 234
alter program semantics 199
appending macro definitions, preprocessed output 87
architecture 123
  -mtune compiler option 125
  -qarch compiler option 123
  -qcache compiler option 130
  -qtune compiler option 125
  macros 269
arrays
  padding 145

B

basic example, described xiii
built-in functions 273, 439
  BCD 290
  Binary-coded decimal 290
    __bcd_invalid 292
    __bcdadd 291
    __bcdadd_ofl 292
    __bcdcmpeq 292
    __bcdcmpge 292
    __bcdcmpgt 292
    __bcdcmple 293
    __bcdcmplt 293
    __bcdsub 291
    __bcdsub_ofl 292
    vec_ldrmb 293
    vec_strmb 293
  block-related 309
  cache-related 301
  cryptography 303
    __vcipher 304
    __vcipherlast 304
    __vncipher 304
built-in functions *(continued)*
  cryptography *(continued)*
    __vncipherlast 305
    __vpermxor 307
    __vpmsumb 307
    __vpmsumd 308
    __vpmsumh 308
    __vpmsumw 308
    __vsbox 305
    __vshasigmad 306
    __vshasigmaw 306
  fixed-point 273
  floating-point 281
  GCC atomic memory access 430
  miscellaneous 442
  synchronization and atomic 294
  transactional memory 447

C

cleanpdf command 173
compatibility
  compatibility
    options for compatibility 59
compiler options 5
  performance optimization 56
  resolving conflicts 6
  specifying compiler options 5
    command line 5
    configuration file 5
    source files 6
  summary of command line options 47
compiler predefined macros 263
configuration 37
  custom configuration files 37
  specifying compiler options 5
configuration file 72
configuration file attributes
  overview 42
configuration files
  code attributes 42
  customizing 42
  editing 42
  ipa attributes 43
  options attributes 43
control of transformations 199

D

data types 122
  -qaltivec compiler option 122
defaults
  customizing 42

E

edits
  configuration files 42
environment variables
  compile-time and link-time 18
  OpenMP
    OMP_DYNAMIC 25
    OMP_PLACES 29
    OMP_PROC_BIND 31
    OMP_STACKSIZE 35
    OMP_THREAD_LIMIT 36
    OMP_WAIT_POLICY 37
  runtime
    XLSMPOPTS 20
  scheduling algorithm environment variable 35
  setting 17
  XLSMPOPTS environment variable 19
error checking and debugging 52
  -g compiler option 112
  -qcheck compiler option 133
  -qlinedebug compiler option 161
exception handling
  for floating point 103

F

floating-point
  exceptions 103

G

GCC 439
GCC options 222

H

high order transformation 145

I

implicit timestamps 204
inlining 93
interprocedural analysis (IPA) 152
invocations 1
  compiler or components 1
  preprocessor 7
  selecting 1
  syntax 2

L

language level 212
language standards 212
lib*.a library files 121
lib*.so library files 121
libraries
  redistributable 11
  XL C/C++ 11
linker 9
  invoking 9

© Copyright IBM Corp. 1996, 2015

linking 9
options that control linking 58
order of linking 10
listing 13
-qlist compiler option 162
options that control listings and messages 55

M

macro definitions, preprocessed output 87
macros
related to architecture 269
related to compiler options 267
related to language features 269
related to the compiler 264
related to the platform 266
maf suboption of -qfloat 202
mergepdf 173

O

object size checking 439
OMP_DISPLAY_ENV environment variable 24
OMP_DYNAMIC environment variable 25
OMP_MAX_ACTIVE_LEVELS 27
OMP_NESTED environment variable 27
OMP_NUM_THREADS environment variable 28
OMP_PLACES environment variable 29
OMP_PROC_BIND environment variable 31
OMP_SCHEDULE environment variable 35
OMP_STACKSIZE environment variable 35
OMP_THREAD_LIMIT environment variable 36
OMP_WAIT_POLICY environment variable 37
OpenMP 24
OpenMP environment variables 24, 35, 37
optimization 56
-O compiler option 76
-qalias compiler option 99
-qoptimize compiler option 76
controlling, using option_override pragma 233
loop optimization 56
-qhot compiler option 145
-qstrict_induction compiler option 204
options for performance optimization 56

P

parallel processing 24
OpenMP environment variables 24
parallel processing pragmas 242
pragma directives 242
parallel processing *(continued)*
setting parallel processing environment variables 19
performance 56
-O compiler option 76
-qalias compiler option 99
-qoptimize compiler option 76
pragmas 228
nosimd 232
unroll 240
profile-directed feedback (PDF) 170
-qpdf1 compiler option 170
-qpdf2 compiler option 170
profiling 128
-qpdf1 compiler option 170
-qpdf2 compiler option 170
-qshowpdf compiler option 189

R

rrm suboption of -qfloat 202

S

shared objects 209
-shared (-qmkshrobj) 209
shared-memory parallelism (SMP) 20
-qsmp compiler option 193
environment variables 20
showpdf 173
SIGTRAP signal 103

T

target machine 123
templates
-qtmplinst compiler option 205
transformations, control of 199
tuning 125
-march compiler option 125
-mtune compiler option 125
-qarch compiler option 125
-qtune compiler option 125

V

vector built-in functions
vec_abs 310
vec_abss 310
vec_add 311
vec_add_u128 313
vec_addc 312
vec_addc_u128 313
vec_adde_u128 314
vec_addec_u128 314
vec_adds 312
vec_all_in 318
vec_and 325
vec_andc 326
vec_any_out 338
vec_avg 338
vec_bperm 339
vec_ceil 339
vec_cipher_be 340
vec_cipherlast_be 340
vector built-in functions *(continued)*
vec_cmpb 340
vec_cmpeq 341
vec_cmpgt 343
vec_cmplt 345
vec_cntlz 345
vec_cpsgn 346
vec_dss 350
vec_dssall 351
vec_dst 351
vec_dstst 351
vec_dststt 352
vec_dstt 352
vec_eqv 353
vec_expte 354
vec_extract 355
vec_floor 355
vec_gbb 356
vec_insert 356
vec_ld 357
vec_lde 358
vec_ldl 359
vec_loge 360
vec_lvsl 361
vec_lvsr 361
vec_madd 362
vec_madds 363
vec_mergee 364
vec_mergeo 366
vec_mfvscr 367
vec_mladd 368
vec_mradds 369
vec_msum 370
vec_msums 371
vec_mtvscr 371
vec_mul 372
vec_mule 372
vec_mulo 373
vec_nabs 374
vec_nand 374
vec_ncipher_be 375
vec_ncipherlast_be 376
vec_nearbyint 376
vec_neg 377
vec_nor 378
vec_orc 381
vec_pack 382
vec_packpx 383
vec_packs 383
vec_packsu 384
vec_perm 384
vec_pmsum_be 385
vec_popcnt 386
vec_recipdiv 388
vec_revb 388
vec_reve 389
vec_rl 390
vec_round 391
vec_rsqrt 393
vec_sbox_be 394
vec_shasigma_be 397
vec_sl 397
vec_sld 398
vec_sldw 399
vec_sll 400
vec_slo 400
vec_splat 401

vector built-in functions *(continued)*
vec_splat_s16 403
vec_splat_s32 403
vec_splat_s8 402
vec_splat_u16 404
vec_splat_u32 405
vec_splat_u8 404
vec_splats 402
vec_sr 406
vec_sra 406
vec_srl 407
vec_sro 408
vec_st 408
vec_ste 409
vec_stl 410
vec_sub_u128 412
vec_subc 412
vec_subc_u128 413
vec_sube_u128 413
vec_subec_u128 414
vec_subs 414
vec_sum2s 415
vec_sum4s 415
vec_sums 416
vec_trunc 416
vec_unpackh 416
vec_unpackl 417
vec_vclz 417
vec_vgbbd 418
vector data types 122
-qaltivec compiler option 122
vector processing 190
-qaltivec compiler option 122
virtual function table (VFT) 92
-fdump-class-hierarchy (-qdump_class_hierarchy) 92
visibility attributes 110
VMX built-in functions
vec_xl 419
vec_xl_be 421
vec_xst 426
vec_xst_be 427

X

XLSMPOPTS environment variable 20

Product Number: 5765-J08; 5725-C73

Printed in USA

SC27-6570-02