IBM XL C for AIX, V13.1.3

Compiler Reference

Version 13.1.3

SC27-4239-02

IBM XL C for AIX, V13.1.3

Compiler Reference

Version 13.1.3

SC27-4239-02

Note

Before using this information and the product it supports, read the information in “Notices” on page 631.

First edition

This edition applies to IBM XL C for AIX, V13.1.3 (Program 5765-J06; 5725-C71) and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

Contents

About this 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 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 7
Specifying compiler options in program source files 7
Resolving conflicting compiler options. 8
Specifying compiler options for architecture-specific compilation 9
Reusing GNU C compiler options with gxlc . . . 11
gxlc syntax 11
Preprocessing. 12
Directory search sequence for included files . . 12
Linking. 13
Order of linking 14
Redistributable libraries 14
Compatibility with earlier versions 15
Compiler messages and listings. 16
Compiler messages 16
Compiler return codes. 18
Compiler listings 19
Message catalog errors. 21
Paging space errors during compilation 22

Chapter 2. Configuring compiler defaults 23
Setting environment variables 23
Compile-time and link-time environment variables 24
Runtime environment variables. 24
Environment variables for parallel processing . . 25
Using custom compiler configuration files 38
Creating custom configuration files 39
Configuring the gxlc option mapping. 42

Chapter 3. Tracking and reporting compiler usage 45
Understanding utilization tracking and reporting. . 45
Overview 45
Four usage scenarios 46
Preparing to use this feature. 54
Time synchronization 54
License types and user information 54
Central configuration 55
Concurrent user considerations. 55
Usage file considerations 56
Regular utilization checking 58
Testing utilization tracking 58
Configuring utilization tracking 60
Editing utilization tracking configuration file entries 60
Understanding the utilization reporting tool . . . 64
Utilization reporting tool command-line options 64
Generating usage reports 68
Understanding usage reports 68
Pruning usage files 71
Diagnostic messages from utilization tracking and reporting 72
Tracking compiler usage with Software License Metric Tags logging 72

Chapter 4. Compiler options reference 75
Summary of compiler options by functional category 75
Output control 75
Input control 76
Language element control 77
Floating-point and integer control 78
Object code control 79
Error checking and debugging 81
Listings, messages, and compiler information . . 84
Optimization and tuning 85
Linking. 89
Portability and migration. 90
Compiler customization 91
Deprecated options 91
Individual option descriptions 92
-# (pound sign) 93
-q32, -q64 94
-qaggrcopy 95
-qalias 96
-qalign 98
-qalloca, -ma 100
-qaltivec 101
-qarch 102
-qasm 105
-qasm_as 107
-qassert 108
-qattr 108
-b 109

-B 110
-qbitfields. 111
-bmaxdata 112
-brtl 113
-c 114
-C, -C! 115
-qcache 116
-qchars 118
-qcheck 119
-qcompact 122
-qconcurrentupdate 123
-qcpluscmt 123
-qcrt 124
-qc_stdinc 125
-D 126
-qdataimported, -qdatalocal, -qtocdata 127
-qdbgfmt 129
-qdbxextra 130
-qdfp 131
-qdigraph 132
-qdirectstorage 133
-qdollar 133
-qdpcl 134
-e 135
-E 136
-qenum 137
-qexpfile 141
-qextchk 141
-f 142
-F 143
-qfdpr 144
-qflag 145
-qfloat 146
-qflttrap 151
-qformat 155
-qfullpath 156
-qfuncsect 157
-qfunctrace 158
-g 160
-G 163
-qgenproto 164
-qhalt 165
-qhaltonmsg 166
-qheapdebug 167
-qhelp 169
-qhot 169
-I 172
-qidirfirst 173
-qignerrno 174
-qignprag. 175
-qinclude 176
-qinfo 178
-qinitauto. 186
-qinlglue 188
-qinline 189
-qipa 193
-qisolated_call 199
-qkeepparm 202
-qkeyword 203
-l 204
-L 205
-qlanglvl 206
-qlargepage 211
-qldbl128, -qlongdouble 212
-qlib 213
-qlibansi 214
-qlibmpi 215
-qlinedebug 216
-qlist 217
-qlistfmt 218
-qlistopt 221
-qlonglit 222
-qlonglong 223
-ma. 224
-qmacpstr 224
-qmakedep, -M 226
-qmaxerr 228
-qmaxmem 229
-qmbcs, -qdbcs 230
-MF 231
-qminimaltoc 232
-qmkshrobj 233
-o 235
-O, -qoptimize 236
-qoptdebug 239
-qoptfile 241
-p, -pg, -qprofile 243
-P 244
-qpath 245
-qpdf1, -qpdf2 247
-qphsinfo 253
-qpic 254
-qppline 255
-qprefetch 256
-qprint 259
-qprocimported, -qproclocal, -qprocunknown 260
-qproto 262
-r 263
-qreport 263
-qreserved_reg 265
-qrestrict 266
-qro 267
-qroconst 268
-qroptr 270
-s 271
-S 271
-qsaveopt. 272
-qshowinc 275
-qshowmacros 276
-qshowpdf 277
-qsimd 278
-qskipsrc 280
-qsmallstack 281
-qsmp 281
-qsource 286
-qsourcetype. 287
-qspeculateabsolutes 288
-qspill 289
-qsrcmsg 290
-qstackprotect 291
-qstatsym. 292
-qstdinc 292

-qstrict 294
-qstrict_induction 299
-qsuppress 299
-qsymtab 301
-qsyntaxonly 302
-t 303
-qtabsize 304
-qtbtable 305
-qthreaded 306
-qtimestamps 307
-qtls 307
-qtocmerge 309
-qtrigraph 310
-qtune 310
-U 313
-qunroll 314
-qunwind. 317
-qupconv 318
-qutf 319
-v, -V 319
-qvecnvol. 320
-qversion 321
-qvisibility 323
-w 325
-W 326
-qwarn64 327
-qweakexp 328
-qweaksymbol 329
-qxcall 330
-qxref 330
-y 332
-Z 333

Chapter 5. Compiler pragmas reference 335

Pragma directive syntax 335
Scope of pragma directives 336
Summary of compiler pragmas by functional category 336
Language element control 336
Floating-point and integer control 337
Error checking and debugging. 337
Optimization and tuning 337
Object code control 338
Portability and migration 339
Deprecated directives. 339
Individual pragma descriptions 339
#pragma align 340
#pragma alloca 340
#pragma block_loop 340
#pragma chars 343
#pragma comment. 343
#pragma disjoint 345
#pragma enum 346
#pragma execution_frequency 346
#pragma expected_value 348
#pragma fini 349
#pragma GCC visibility push, #pragma GCC visibility pop 349
#pragma ibm independent_loop 351
#pragma ibm iterations 352
#pragma ibm max_iterations 353
#pragma ibm min_iterations 354
#pragma ibm snapshot 355
#pragma info 355
#pragma init 355
#pragma isolated_call 356
#pragma langlvl 356
#pragma leaves. 356
#pragma loopid 357
#pragma map 358
#pragma mc_func 359
#pragma nofunctrace 361
#pragma nosimd 362
#pragma novector 362
#pragma options 362
#pragma option_override 364
#pragma pack 366
#pragma reachable 369
#pragma reg_killed_by 370
#pragma simd_level 372
#pragma STDC CX_LIMITED_RANGE 373
#pragma stream_unroll 374
#pragma strings 375
#pragma unroll, #pragma nounroll 375
#pragma unrollandfuse 375
#pragma weak 377
Pragma directives for parallel processing . . . 380

Chapter 6. Compiler predefined macros 403

General macros. 403
Macros indicating the XL C compiler 404
Macros related to the platform 405
Macros related to compiler features 405
Macros related to compiler option settings. . . 406
Macros related to architecture settings 407
Macros related to language levels 408

Chapter 7. Compiler built-in functions 413

Fixed-point built-in functions 413
Absolute value functions 413
Assert functions 414
Bit permutation functions 414
Comparison functions 414
Count zero functions 415
Division functions 415
Load functions 417
Multiply functions. 417
Population count functions 418
Rotate functions 419
Store functions 420
Trap functions 421
Binary floating-point built-in functions 422
Absolute value functions 422
Add functions 422
Conversion functions 423
FPSCR functions 425
Multiply functions. 428
Multiply-add/subtract functions 428
Reciprocal estimate functions 429

Rounding functions 430
Select functions. 431
Square root functions. 431
Software division functions. 432
Store functions 433
Binary-coded decimal built-in functions 433
BCD add and subtract 433
BCD test add and subtract for overflow . . . 434
BCD comparison 435
BCD load and store 436
Decimal floating-point built-in functions 437
Absolute value functions 437
Coefficient functions 438
Comparison functions 439
Conversion functions. 440
Exponent functions 445
NaN functions 446
Register transfer functions 447
Rounding functions 448
Test functions 450
Miscellaneous functions 455
Synchronization and atomic built-in functions . . 456
Check lock functions 456
Clear lock functions 457
Compare and swap functions 458
Fetch functions 459
Load functions 461
Store functions 462
Synchronization functions 462
Cache-related built-in functions 464
Data cache functions 464
Prefetch built-in functions 466
Cryptography built-in functions 474
Advanced Encryption Standard functions . . . 474
Secure Hash Algorithm functions. 476
Miscellaneous functions 477
Block-related built-in functions 479
__bcopy 479
bzero 480
Vector built-in functions. 480
vec_abs 481
vec_abss 481
vec_add 482
vec_addc 482
vec_adds 483
vec_add_u128 484
vec_addc_u128 484
vec_adde_u128 484
vec_addec_u128 485
vec_all_eq 485
vec_all_ge 486
vec_all_gt 488
vec_all_in 489
vec_all_le. 489
vec_all_lt 490
vec_all_nan 491
vec_all_ne 492
vec_all_nge 493
vec_all_ngt 494
vec_all_nle 494
vec_all_nlt 495
vec_all_numeric 495
vec_and 496
vec_andc 497
vec_any_eq 498
vec_any_ge 499
vec_any_gt 501
vec_any_le 502
vec_any_lt 503
vec_any_nan 504
vec_any_ne 505
vec_any_nge. 506
vec_any_ngt 507
vec_any_nle 507
vec_any_nlt 508
vec_any_numeric 508
vec_any_out. 509
vec_avg 509
vec_bperm 510
vec_ceil 510
vec_cmpb. 511
vec_cmpeq 511
vec_cmpge 512
vec_cmpgt 513
vec_cmple 514
vec_cmplt 515
vec_cntlz 516
vec_cpsgn 516
vec_ctd 517
vec_ctf 517
vec_cts 518
vec_ctsl 518
vec_ctu 519
vec_ctul 519
vec_cvf 520
vec_div 520
vec_dss 521
vec_dssall 521
vec_dst 521
vec_dstst 522
vec_dststt 522
vec_dstt 523
vec_eqv 523
vec_expte. 525
vec_extract 525
vec_floor 526
vec_gbb 526
vec_insert 527
vec_ld 528
vec_lde 529
vec_ldl 530
vec_loge 531
vec_lvsl 532
vec_lvsr 532
vec_madd 533
vec_madds 534
vec_max 534
vec_mergee 535
vec_mergeh 536
vec_mergel 537
vec_mergeo 538
vec_mfvscr 539

vec_min 539
vec_mladd 540
vec_mradds 541
vec_msub 541
vec_msum 542
vec_msums 543
vec_mtvscr 543
vec_mul 544
vec_mule 544
vec_mulo 545
vec_nabs 546
vec_nand 546
vec_neg 548
vec_nmadd 548
vec_nmsub 549
vec_nor 549
vec_or 550
vec_orc 552
vec_pack 553
vec_packpx 554
vec_packs 554
vec_packsu 555
vec_perm. 556
vec_permi 556
vec_popcnt 557
vec_promote. 558
vec_re 559
vec_revb 559
vec_reve 560
vec_rl 561
vec_round 561
vec_roundc 562
vec_roundm 562
vec_roundp 563
vec_roundz 563
vec_rsqrte 564
vec_sel 564
vec_sl 565
vec_sld 566
vec_sldw 567
vec_sll. 568
vec_slo 569
vec_splat 570
vec_splats 570
vec_splat_s8 571
vec_splat_s16 571
vec_splat_s32 572
vec_splat_u8. 572
vec_splat_u16 573
vec_splat_u32 573
vec_sqrt 574
vec_sr 574
vec_sra 575
vec_srl 576
vec_sro 577
vec_st 577
vec_ste 579
vec_stl. 580
vec_sub 581
vec_sub_u128 582
vec_subc 583
vec_subc_u128 583
vec_sube_u128 583
vec_subec_u128 584
vec_subs 584
vec_sum2s 585
vec_sum4s 585
vec_sums. 586
vec_trunc. 586
vec_unpackh 586
vec_unpackl 587
vec_xl 588
vec_xl_be. 589
vec_xld2 590
vec_xlds 591
vec_xlw4 592
vec_xor 593
vec_xst 594
vec_xst_be 595
vec_xstd2. 596
vec_xstw4 597
GCC atomic memory access built-in functions (IBM extension) 598
Atomic lock, release, and synchronize functions 599
Atomic fetch and operation functions 600
Atomic operation and fetch functions 603
Atomic compare and swap functions 606
Miscellaneous built-in functions 607
Optimization-related functions 607
Move to/from register functions 608
Memory-related functions 610
Built-in functions for parallel processing 612
IBM SMP built-in functions. 613
Transactional memory built-in functions . . . 613

Chapter 8. OpenMP runtime functions for parallel processing 621
omp_get_max_active_levels 621
omp_set_max_active_levels. 621
omp_get_schedule. 622
omp_set_schedule 622
omp_get_thread_limit 623
omp_get_level 623
omp_get_ancestor_thread_num 623
omp_get_team_size 623
omp_get_active_level. 624
omp_get_num_threads 624
omp_set_num_threads 624
omp_get_max_threads 625
omp_get_thread_num 625
omp_get_num_procs 625
omp_in_final 625
omp_in_parallel 625
omp_set_dynamic 626
omp_get_dynamic. 626
omp_set_nested 626
omp_get_nested 627
omp_init_lock, omp_init_nest_lock 627
omp_destroy_lock, omp_destroy_nest_lock . . . 627
omp_set_lock, omp_set_nest_lock. 627
omp_unset_lock, omp_unset_nest_lock 628
omp_test_lock, omp_test_nest_lock 628

omp_get_wtime 628
omp_get_wtick 629

Notices 631
Trademarks 633

Index 635

About this document

This document is a reference for the IBM® XL C for AIX®, V13.1.3 compiler. Although it provides information about compiling and linking applications written in 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 developers who have some familiarity with the XL C compilers or other command-line compilers on AIX operating systems. It assumes thorough knowledge of the C programming language and basic knowledge of operating system commands. Although this information is intended as a reference guide, programmers new to XL C can still find information about the capabilities and features unique to the XL C compiler.

How to use this document

Throughout this document, the **xlc** command invocation is 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 applications using the XL C compiler, it does not include the following topics:

- Compiler installation: see the *XL C Installation Guide*.
- The C programming language: see the *XL C Language Reference* for information about the syntax, semantics, and IBM implementation of the C programming language.
- Programming topics: see the *XL C Optimization and Programming Guide* for detailed information about developing applications with XL 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; how to reuse GNU C compiler options through the use of the compiler utility **gxlc**; and compiler listings and messages.

Chapter 2, “Configuring compiler defaults,” on page 23 discusses topics related to setting up default compilation settings, including setting environment variables, customizing the configuration file, and customizing the **gxlc** option mappings.

Chapter 3, “Tracking and reporting compiler 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 75 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 each compiler option sorted alphabetically.

Chapter 5, “Compiler pragmas reference,” on page 335 provides a summary of pragma directives according to their functional category, which allows you to look up and link to pragmas by function. This chapter includes individual descriptions of pragmas sorted alphabetically, including OpenMP and SMP directives.

Chapter 6, “Compiler predefined macros,” on page 403 provides a list of compiler macros grouped according to their category.

Chapter 7, “Compiler built-in functions,” on page 413 contains individual descriptions of XL C built-in functions for Power® architectures, categorized by their functionality.

Chapter 8, “OpenMP runtime functions for parallel processing,” on page 621 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 for AIX, 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**, along with several other compiler invocation commands to support various C language levels and compilation environments.
italics	Parameters or variables whose actual names or values are to be supplied by the user. Italics are also used to introduce new terms.	Make sure that you update the *size* parameter if you return more than the *size* requested.
<u>underlining</u>	The default setting of a parameter of a compiler option or directive.	nomaf | <u>maf</u>
`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)

In descriptions of language elements where a feature is exclusive to the C11 standard, or where a feature is an IBM extension of the C standard, this information uses icons to delineate segments of text as follows:

Table 2. Qualifying elements

Qualifier/Icon	Meaning
IBM extension begins ▶ IBM IBM ◀ IBM extension ends	The text describes a feature that is an IBM extension to the standard language specifications.
C11 begins ▶ C11 C11 ◀ C11 ends	The text describes a feature that is introduced into standard C as part of C11.

Syntax diagrams

Throughout this information, diagrams illustrate XL C syntax. This section helps you to interpret and use those diagrams.

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

  The ▶▶── symbol indicates the beginning of a command, directive, or statement.

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

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

  The ──▶◀ symbol indicates the end of a command, directive, or statement.

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

```
▶▶──keyword──required_argument─────────────────────────────────────────────▶◀
```

- Optional items are shown below the main path:

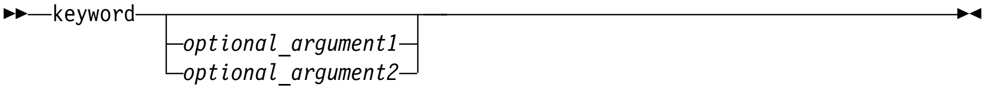
```
▶▶──keyword──┬───────────────────┬─────────────────────────────────────────▶◀
             └─optional_argument─┘
```

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

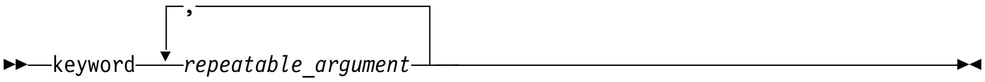
  If you *must* choose one of the items, one item of the stack is shown on the main path.

```
▶▶──keyword──┬─required_argument1─┬────────────────────────────────────────▶◀
             └─required_argument2─┘
```

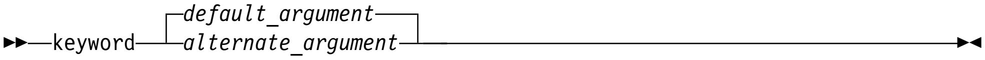
If choosing one of the items is optional, the entire stack is shown below the main path.



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



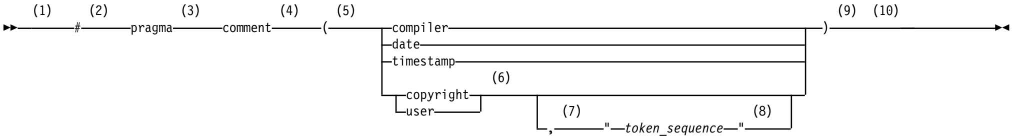
- The item that is the default is shown above the main path.



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

Sample syntax diagram

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



Notes:

1 This is the start of the syntax diagram.

2 The symbol # must appear first.

3 The keyword pragma must appear following the # symbol.

4 The name of the pragma comment must appear following the keyword pragma.

5 An opening parenthesis must be present.

6 The comment type must be entered only as one of the types indicated: compiler, date, timestamp, copyright, or user.

7 A comma must appear between the comment type copyright or user, and an optional character string.

8 A character string must follow the comma. The character string must be enclosed in double quotation marks.

9 A closing parenthesis is required.

10 This is the end of the syntax diagram.

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

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

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:

IBM XL C information

XL 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 for AIX, V13.1.3. It is located by default in the XL 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 directory and in the root directory 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 for AIX, 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/SSGH2K_13.1.3/com.ibm.compilers.aix.doc/welcome.html.
- PDF documents

  PDF documents are available on the web at http://www.ibm.com/support/docview.wss?uid=swg27036590.

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

Table 3. XL C PDF files

Document title	PDF file name	Description
IBM XL C for AIX, V13.1.3 Installation Guide, SC27-4238-02	`install.pdf`	Contains information for installing XL C and configuring your environment for basic compilation and program execution.
Getting Started with IBM XL C for AIX, V13.1.3, SC27-4237-02	`getstart.pdf`	Contains an introduction to the XL C product, with information about setting up and configuring your environment, compiling and linking programs, and troubleshooting compilation errors.
IBM XL C for AIX, V13.1.3 Compiler Reference, SC27-4239-02	`compiler.pdf`	Contains information about the various compiler options, pragmas, macros, environment variables, and built-in functions, including those used for parallel processing.
IBM XL C for AIX, V13.1.3 Language Reference, SC27-4240-02	`langref.pdf`	Contains information about the C programming languages, as supported by IBM, including language extensions for portability and conformance to nonproprietary standards.
IBM XL C for AIX, V13.1.3 Optimization and Programming Guide, SC27-4241-02	`proguide.pdf`	Contains information about advanced programming topics, such as application porting, interlanguage calls with Fortran code, library development, application optimization and parallelization, and the XL 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, including IBM Redbooks® publications, white papers, and other articles, is available on the web at http://www.ibm.com/support/docview.wss?uid=swg27036590.

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 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*. (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.
- *Information Technology - Programming Languages - Extension for the programming language C to support decimal floating-point arithmetic, ISO/IEC WDTR 24732*. This draft technical report has been submitted to the C standards committee, and is available at http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1176.pdf.
- *ANSI/IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-1985*.
- *OpenMP Application Program Interface Version 3.1* (full support), and *OpenMP Application Program Interface Version 4.0 (partial support)*, available at http://www.openmp.org

Other IBM information

- *Parallel Environment for AIX: Operation and Use*
- The IBM Systems Information Center, at http://publib.boulder.ibm.com/infocenter/systems/index.jsp?topic=/com.ibm.aix.doc/doc/base/aixparent.htm, is a resource for AIX information.

  You can find the following books for your specific AIX system:
  - *AIX Commands Reference, Volumes 1 - 6*
  - *Technical Reference: Base Operating System and Extensions, Volumes 1 & 2*
  - *AIX National Language Support Guide and Reference*
  - *AIX General Programming Concepts: Writing and Debugging Programs*
  - *AIX Assembler Language Reference*

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 Support page at http://www.ibm.com/support/entry/portal/product/rational/xl_c_for_aix. 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@ca.ibm.com.

For the latest information about XL C, visit the product information site at http://www.ibm.com/software/products/en/xlcaix.

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 information, send your comments to compinfo@ca.ibm.com.

Be sure to include the name of the manual, the part number of the manual, the version of XL 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 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 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
- "Reusing GNU C compiler options with gxlc" on page 11
- "Preprocessing" on page 12
- "Linking" on page 13
- "Compiler messages and listings" on page 16

Invoking the compiler

Different forms of the XL C compiler invocation commands support various levels of the C language. In most cases, you can use the **xlc** command to compile C source files.

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

Note: For each invocation command, the compiler configuration file defines default option settings and, in some cases, macros; for information about the defaults implied by a particular invocation, see the /opt/IBM/xlc/13.1.3/etc/xlc.cfg file for your system.

Table 4. Compiler invocations

Basic invocations	Description	Equivalent special 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, xlc_r7, xlc128, xlc128_r, xlc128_r4, xlc128_r7
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, c99_r4, c99_r7, c99_128, c99_128_r, c99_128_r4, c99_128_r7

Table 4. Compiler invocations (continued)

Basic invocations	Description	Equivalent special 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, c89_r4, c89_r7, c89_128, c89_128_r, c89_128_r4, c89_128_r7
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, cc_r4, cc_r7, cc128, cc128_r, cc128_r4, cc128_r7
gxlc	Invokes the compiler for C source files. This command accepts many common GNU C options, maps them to their XL C option equivalents, and then invokes **xlc**. For more information, see "Reusing GNU C compiler options with gxlc" on page 11.	

Table 5. Suffixes for special invocations

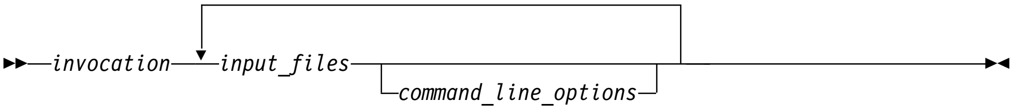
128-suffixed invocations	All **128**-suffixed invocation commands are functionally similar to their corresponding base compiler invocations. They specify the **-qldbl128** option, which increases the length of `long double` types in your program from 64 to 128 bits. They also link with the 128-bit versions of the C runtime libraries.
_r-suffixed invocations	All **_r**-suffixed invocations allow for threadsafe compilation and you can use them to link the programs that use multithreading. Use these commands if you want to create threaded applications. The **_r7** invocations are provided to help migrate programs based on Posix Draft 7 to Posix Draft 10. The **_r4** invocations should be used for DCE threaded applications. For more information about DCE, see What is DCE? in the CICS® Transaction Server for z/OS® Information Center.

Related information

- "-qlanglvl" on page 206

Command-line syntax

You invoke the compiler using the following syntax, where *invocation* can be replaced with any valid XL 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 12 for details.

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

C source files

These are files containing C source code.

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

Preprocessed source files

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

Object files

Object files must have a .o suffix, for example, `*file_name*.o`. Object files, library files, and unstripped executable files serve as input to the linker. After compilation, the linker links all of the specified object files to create an executable file.

Assembler files

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

Unpreprocessed assembler files

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

Shared library files

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

Unstripped executable files

Extended Common Object File Format (XCOFF) files that have not been stripped with the operating system **strip** command can be used as input to the compiler. See the **strip** command in the *AIX Commands Reference* and the description of a.out file format in the *AIX Files Reference* for more information.

Related information:

"Input control" on page 76

Types of output files

You can specify the following types of output files when invoking the XL 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.

The format of the a.out file is described in the *AIX Files Reference*.

Object files

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

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

Shared library files

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

Assembler files

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

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

Preprocessed source files

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

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

Listing files

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

Target files
: If you specify the **-qmakedep** or **-M** option, a target file suitable for inclusion in a makefile, *file_name*.u is produced for each input file. You can use the **-MF** option to specify the name or location for the dependency output files that are generated by the **-qmakedep** or **-M** option.

Related information:

"Output control" on page 75

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 compiler resolves most of these conflicts and incompatibilities in a consistent fashion, as follows:

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

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

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

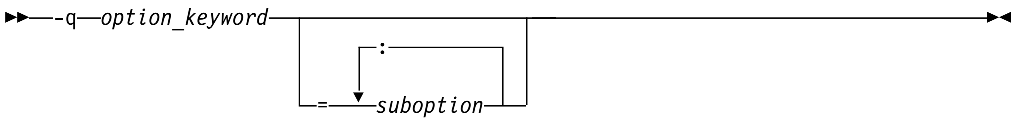
Specifying compiler options on the command line

Most options specified on the command line override both the default settings of the option and options set in the configuration file. Similarly, most options specified on the command line are in turn overridden by pragma directives, which provide you a means of setting compiler options right in the source file. Options that do not follow this scheme are listed in "Resolving conflicting compiler options" on page 8.

There are two kinds of command-line options:

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

-q options



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

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

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

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

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

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

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

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

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

Flag options

XL C supports a number of common conventional flag options used on UNIX systems. Lowercase flags are different from their corresponding uppercase flags. For example, **-c** and **-C** are two different compiler options: **-c** specifies that the compiler should only preprocess and compile and not invoke the linker, while **-C** can be used with **-P** or **-E** to specify that user comments should be preserved.

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

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

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

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

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

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

has the same effect as:

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

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

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

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

has the same effect as:

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

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

Specifying compiler options in a configuration file

The default configuration file (/opt/IBM/xlc/13.1.3/etc/xlc.cfg) defines values and compiler options for the compiler. The compiler refers to this file when compiling 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 38.

Specifying compiler options in program source files

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

- Using **#pragma options** ***option_name*** syntax

  You can use command-line options with the **#pragma options** syntax, which takes the same name as the option, and suboptions with a syntax identical to that of the option. For example, if the command-line option is:

  ```
  -qhalt=w
  ```

  The pragma form is:

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

  The descriptions for each individual option indicates whether this form of the pragma is supported. For details, see “#pragma options” on page 362.

- 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 92, 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.

Complete details on pragma syntax are provided in "Pragma directive syntax" on page 335.

Other pragmas do not have equivalent command-line options; these are described in detail throughout Chapter 5, "Compiler pragmas reference," on page 335.

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

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

Resolving conflicting compiler options

In general, if more than one variation of the same option is specified (with the exception of **-qxref** and **-qattr**), the compiler uses the setting of the last one specified. Compiler options specified on the command line must appear in the order you want the compiler to process them. However, some options have cumulative effects when they are specified more than once; examples are the **-I***directory* and **-L***directory* 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. Rules for resolving conflicts between compiler mode and architecture-specific options are discussed in "Specifying compiler options for architecture-specific compilation."

Option	Conflicting options	Resolution
-qalias=allptrs	-qalias=noansi	-qalias=noansi
-qalias=typeptr	-qalias=noansi	-qalias=noansi
-qhalt	Multiple severities specified by **-qhalt**	Lowest severity specified
-qnoprint	-qxref, -qattr, -qsource, -qlistopt, -qlist	-qnoprint
-qfloat=rsqrt	-qnoignerrno	Last option specified
-qxref	-qxref=full	-qxref=full
-qattr	-qattr=full	-qattr=full
-qfloat=hsflt	-qfloat=spnans	-qfloat=hsflt
-qfloat=hssngl	-qfloat=spnans	-qfloat=hssngl
-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
-qnostdinc	-qc_stdinc	-qnostdinc

Specifying compiler options for architecture-specific compilation

You can use the **-q32**, **-q64**, **-qarch**, and **-qtune** compiler options to optimize the output of the compiler to suit:

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

Generally speaking, the options do the following:

- **-q32** selects 32-bit execution mode.
- **-q64** selects 64-bit execution mode.
- **-qarch** selects the general family processor architecture for which instruction code should be generated. Certain **-qarch** settings produce code that will run *only* on systems that support *all* of the instructions generated by the compiler in response to a chosen **-qarch** setting.
- **-qtune** selects the specific processor for which compiler output is optimized. Some **-qtune** settings can also be specified as **-qarch** options, in which case they do not also need to be specified as a **-qtune** option. The **-qtune** option influences only the performance of the code when running on a particular system but does not determine where the code will run.

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

1. Internal default (32-bit mode)

2. OBJECT_MODE environment variable setting
3. Configuration file settings
4. Command line compiler options (**-q32**, **-q64**, **-qarch**, and **-qtune**)
5. Source file statements (**#pragma options tune=***suboption*)

The compilation mode actually used by the compiler depends on a combination of the settings of the **-q32**, **-q64**, **-qarch**, and **-qtune** compiler options, subject to the following conditions:

- *Compiler mode* is set according to the last-found instance of the **-q32** or **-q64** compiler options. If neither of these compiler options is set, the compiler mode is set by the value of the OBJECT_MODE environment variable. If the OBJECT_MODE environment variable is also not set, the compiler assumes 32-bit compilation mode.
- *Architecture target* is set according to the last-found instance of the **-qarch** compiler option, provided that the specified **-qarch** setting is compatible with the *compiler mode* setting. If the **-qarch** option is not set, the compiler sets **-qarch** to the appropriate default based on the effective compiler mode setting.
- Tuning of the architecture target is set according to the last-found instance of the **-qtune** compiler option, provided that the **-qtune** setting is compatible with the *architecture target* and *compiler mode* settings. If the **-qtune** option is not set, the compiler assumes a default **-qtune** setting according to the **-qarch** setting in use. If **-qarch** is not specified, the compiler sets **-qtune** to the appropriate default based on the effective **-qarch** as selected by default based on the effective compiler mode setting.

Allowable combinations of these options are found in "-qtune" on page 310.

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

- **-q32** or **-q64** setting is incompatible with user-selected **-qarch** option.

  Resolution: **-q32** or **-q64** setting overrides the **-qarch** option; compiler issues a warning message, sets **-qarch** to its default setting, and sets the **-qtune** option accordingly to its default value.
- **-qarch** option is incompatible with user-selected **-qtune** option.

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

  Resolution: Compiler issues a warning message, sets **-qarch** and **-qtune** to their default settings. The compiler mode (32-bit or 64-bit) is determined by the OBJECT_MODE environment variable or **-q32** or **-q64** compiler settings.

Related information

- "-qarch" on page 102
- "-qtune" on page 310
- "-q32, -q64" on page 94

Reusing GNU C compiler options with gxlc

The gxlc utility accepts GNU C compiler options and translates them into comparable XL C options. It uses the XL C options to create an **xlc** invocation command, which the utility uses to invoke XL C. The **gxlc** utility is provided to facilitate the reuse of makefiles created for applications previously developed with GNU C. However, to fully exploit the capabilities of XL C, it is recommended that you use the XL C invocation command **xlc** and its associated options.

The actions of **gxlc** are controlled by the configuration file /opt/IBM/xlc/13.1.3/etc/gxlc.cfg. The GNU C options that have an XL C counterpart are shown in this file. Not every GNU option has a corresponding XL C option. The **gxlc** utility returns a warning for any GNU C option it cannot translate.

The **gxlc** option mappings are modifiable. For information on adding to or editing the **gxlc** configuration file, see "Configuring the gxlc option mapping" on page 42.

gxlc syntax

The following diagram shows the gxlc syntax:

```
►►──gxlc──┬──────┬──┬────────────────────┬──┬───────────────┬─filename─────────────►◄
          ├─-v───┤  └─-Wx,─xlc_options─┘  └─gcc_options─┘
          └─-vv──┘
```

where:

filename
Is the name of the file to be compiled.

-v
Verifies the command that is used to invoke XL C. The utility displays the XL C invocation command that it has created, before using it to invoke the compiler.

-vv
Runs a simulation. The utility displays the XL C invocation command that it has created, but does not invoke the compiler.

-Wx, *xlc_ options*
Sends the given XL C options directly to the **xlc** invocation command. The utility adds the given options to the XL C invocation it is creating, without attempting to translate them. Use this option with known XL C options to improve the performance of the utility. Multiple *xlc_options* are delimited by a comma.

-*gcc_options*
The GNU C options that are translated to XL C options. The utility emits a warning for any option it cannot translate. The GNU C options that are currently recognized by **gxlc** are in the configuration file gxlc.cfg. Multiple *-gcc_options* are delimited by the space character.

Examples

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

```
gxlc -fstrict-aliasing hello.c
```

which translates into:

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

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

Related information

- "Configuring the gxlc option mapping" on page 42

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 136	Preprocesses the source files and writes the output to standard output. By default, `#line` directives are generated.
"-P" on page 244	Preprocesses the source files and creates an intermediary file with a .i file name suffix for each source file. By default, `#line` directives are not generated.
"-qppline" on page 255	Toggles on and off the generation of `#line` directives for the **-E** and **-P** options.
"-C, -C!" on page 115	Preserves comments in preprocessed output.
"-D" on page 126	Defines a macro name from the command line, as if in a `#define` directive.
"-qmakedep, -M" on page 226	Produces the dependency files that are used by the **make** tool for each source file.
"-U" on page 313	Undefines a macro name defined by the compiler or by the **-D** option.
"-qshowmacros" on page 276	Emits macro definitions to preprocessed output.
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 compiler supports the following types of included files:

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

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

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

If you specify the header file using a full (absolute) path name, you can use these methods interchangeably, regardless of the type of header file you want to include. However, if you specify the header file using a *relative* path name, the compiler uses a different directory search order for locating the file depending on the method used to include the file.

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

1. Header files included with **-qinclude** only: The compiler searches the current (working) directory from which the compiler is invoked.[1]
2. Header files included with **-qinclude** or `#include "file_name"`: The compiler searches the directory in which the source file is located.[1]
3. All header files: The compiler searches each directory specified by the **-I** compiler option, in the order that it displays on the command line.
4. All header files: The compiler searches the standard directory for the XL C headers. The default directory for these headers is specified in the compiler configuration file. This is normally /opt/IBM/xlc/13.1.3/include. But the search path can be changed with **-qc_stdinc** compiler option.
5. All header files: The compiler searches the standard directory for the system headers. The default directory for these headers is specified in the compiler configuration file. This is normally /usr/include/. But the search path can be changed with **-qc_stdinc**.

Note:

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

Related information

- "-I" on page 172
- "-qc_stdinc" on page 125
- "-qidirfirst" on page 173
- "-qinclude" on page 176
- "-qstdinc" on page 292

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**
- **-P**

- **-S**
- **-qsyntaxonly**
- **-#**
- **-qhelp**
- **-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`. Library file names have a `.a` or `.so` suffix, for example, *`filename`*`.a`, or *`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 **-qmkshrobj** option to create a shared library, the default name of the shared object created is shr.o. 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 89.

Related information

- "-qmkshrobj" on page 233

Order of linking

The compiler links libraries in the following order:

1. System startup libraries
2. User .o files and libraries
3. XL C libraries
4. C standard libraries

Related information

- "Linking" on page 89
- "Redistributable libraries"
- **ld** in the *AIX Commands Reference, Volume 5: s through u*

Redistributable libraries

If you build your application using XL 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 filesets 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 filesets that contain the redistributable libraries with your application. The filesets are stored under the runtime/ directory on the installation CD.
- 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 support website at http://www.ibm.com/support/entry/portal/product/rational/xl_c_for_aix.

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

Table 6. Redistributable libraries

Fileset	**Libraries (and default installation path)**	**Description**
xlsmp.rte	`/usr/include/omp.h` `/usr/lpp/xlsmp/default_msg/smprt.cat`	SMP runtime environment
xlsmp.aix61.rte	`/usr/lpp/xlsmp/aix61/libxlsmp.a` `/usr/lpp/xlsmp/aix61/libxlomp_ser.a`	SMP runtime libraries for AIX 6.1, AIX 7.1, and AIX 7.2
xlsmp.msg.en_US.rte	`/usr/lib/nls/msg/en_US/smprt.cat`	SMP runtime messages (English, ISO8859-1)
xlsmp.msg.EN_US.rte	`/usr/lib/nls/msg/EN_US/smprt.cat`	SMP runtime messages (English, UTF-8)
xlsmp.msg.ja_JP.rte	`/usr/lib/nls/msg/ja_JP/smprt.cat`	SMP runtime messages (Japanese, IBM-eucJP)
xlsmp.msg.Ja_JP.rte	`/usr/lib/nls/msg/Ja_JP/smprt.cat`	SMP runtime messages (Japanese, IBM-943)
xlsmp.msg.JA_JP.rte	`/usr/lib/nls/msg/JA_JP/smprt.cat`	SMP runtime messages (Japanese, UTF-8)
xlsmp.msg.zh_CN.rte	`/usr/lib/nls/msg/zh_CN/smprt.cat`	SMP runtime messages (Chinese, IBM-eucCN)
xlsmp.msg.ZH_CN.rte	`/usr/lib/nls/msg/ZH_CN/smprt.cat`	SMP runtime messages (Chinese, UTF-8)
xlsmp.msg.Zh_CN.rte	`/usr/lib/nls/msg/Zh_CN/smprt.cat`	SMP runtime messages (Chinese, GBK)
xlccmp.13.1.3.lib	`/opt/IBM/xlc/13.1.3/lib/aix61/libxl.a` `/opt/IBM/xlc/13.1.3/lib/aix61/libxlopt.a`	XL C libraries for AIX 6.1, AIX 7.1, and AIX 7.2

memdbg.adt	`/usr/vac/lib/libhm.a`
	`/usr/vac/lib/libhm_r.a`
	`/usr/vac/lib/libhmd.a`
	`/usr/vac/lib/libhmd_r.a`
	`/usr/vac/lib/libhmu.a`
	`/usr/vac/lib/libhmu_r.a`
	`/usr/vac/lib/libhu.a`
	`/usr/vac/lib/libhu_r.a`
	`/usr/vac/lib/profiled/libhm.a`
	`/usr/vac/lib/profiled/libhm_r.a`
	`/usr/vac/lib/profiled/libhmd.a`
	`/usr/vac/lib/profiled/libhmd_r.a`
	`/usr/vac/lib/profiled/libhmu.a`
	`/usr/vac/lib/profiled/libhmu_r.a`
	`/usr/vac/lib/profiled/libhu.a`
	`/usr/vac/lib/profiled/libhu_r.a`
		User heap/memory debug toolkit

Compatibility with earlier versions

This section describes issues about compatibility with earlier versions and their workarounds.

Compiler option compatibility issues

In IBM XL C for AIX, V13.1.3, the implementation of the threadprivate data, that is, OpenMP threadprivate variable, has been improved. The operating system thread local storage is used instead of the runtime implementation. The new implementation might improve performance on some applications.

If you plan to mix the object files `.o` that you have compiled with levels prior to 11.1 with the object files that you compiled with IBM XL C for AIX, V13.1.3, and the same OpenMP threadprivate variables are referenced in both old and new object files, different implementations might cause incompatibility issues. A link error, a compile time error or other undefined behaviors might occur. To support compatibility with earlier versions, you can use the **-qsmp=noostls** suboption to switch back to the old implementation. You can recompile the entire program with the default suboption **-qsmp=ostls** to get the benefit of the new implementation.

If you are not sure whether the object files you have compiled with levels prior to 11.1 contain any old implementation, you can use the **nm** command to determine whether you need to use the **-qsmp=noostls** suboption. The following code is an example that shows how to use the **nm** command:

```
> nm oldfiles.o
...
._xlGetThStorageBlock U          -
._xlGetThValue        U          -
...
```

In the preceding example, if `_xlGetThStorageBlock` or `_xlGetThValue` is found, this means the object files contain old implementation. In this case, you must use **-qsmp=noostls**; otherwise, use the default suboption **-qsmp=ostls**.

Compiler messages and listings

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

- "Compiler messages"
- "Compiler return codes" on page 18
- "Compiler listings" on page 19
- "Message catalog errors" on page 21
- "Paging space errors during compilation" on page 22

Compiler messages

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

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

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

Related reference:

"-qsource" on page 286

"-qsrcmsg" on page 290

"-qflag" on page 145

"-w" on page 325

"-qinfo" on page 178

Compiler message format

Diagnostic messages have the following format:

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

where

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

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

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

15 Is the compiler product identifier.

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

00	- code generating or optimizing message
01	- compiler services message
05	- message specific to the C compiler
06	- message specific to the C compiler
86	- message specific to interprocedural analysis (IPA)

number
Is the message number.

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

text
Is a message describing the error.

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

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

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

Message severity levels and compiler response

The XL 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 adjust the default compiler response by using any of the following options:

- **-qhalt** halts the compilation phase at a lower severity level than the default.

- **-qmaxerr** halts the compilation phase as soon as a specific number of errors at a specific severity level is reached.
- **-qhaltonmsg** halts the compilation phase as soon as a specific error is encountered.

Table 7. Compiler message severity levels

Letter	**Severity**	**Compiler response**
I	Informational	Compilation continues and object code is generated. The message reports conditions found during compilation.
W	Warning	Compilation continues and object code is generated. The message reports valid but possibly unintended conditions.
E	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	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, the message should be reported to your IBM service representative.

Related information

- "-qhalt" on page 165
- "-qmaxerr" on page 228
- "-qhaltonmsg" on page 166
- "Listings, messages, and compiler information" on page 84

Compiler return codes

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

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

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

Return code	Error type
1	Any error with a severity level higher than the setting of the **-qhalt** compiler option has been detected.
40	An option error or an unrecoverable error has been detected.

41	A configuration file error has been detected.
249	A no-files-specified error has been detected.
250	An out-of-memory error has been detected. The compiler cannot allocate any more memory for its use.
251	A signal-received error has been detected. That is, an unrecoverable error or interrupt signal has occurred.
252	A file-not-found error has been detected.
253	An input/output error has been detected: files cannot be read or written to.
254	A fork error has been detected. A new process cannot be created.
255	An error has been detected while the process was running.

Note: Return codes can also be displayed for runtime errors. For example, a runtime return code of 99 indicates that a static initialization has failed.

gxlc return codes

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

40 A 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. For example, any diagnostic messages emitted during compilation are written to the listing.

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

- -qsource
- -qlistopt
- -qattr
- -qxref
- -qlist
- -qreport

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

Header section
Lists the compiler name, version, release, the source file name, and the date and time of the compilation.

Source section
If you use the **-qsource** option, lists the input source code with line numbers. If there is an error at a line, the associated error message is displayed after the source line. Lines containing macros have additional

lines showing the macro expansion. By default, this section only lists the main source file. Use the **-qshowinc** option to expand all header files as well.

Options section
Lists the options that were in effect during the compilation. By default, it lists the specified options. To get all options, specify the **-qlistopt** option.

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

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

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

Compilation epilogue section

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

Object section

If you 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 84

Message catalog errors

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

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

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

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

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

```
No message text for message_number
```

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

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

```
ls /opt/IBM/xlc/13.1.3/msg/$LANG/*.cat
```

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

The compiler calls the default message catalogs in `/opt/IBM/xlc/13.1.3/exe/default_msg/` when the locale has never been changed from the default, **C**.

- The message catalogs for the locale specified by *LANG* cannot be found.
- The locale has never been changed from the default, **C**.

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

Paging space errors during compilation

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

```
1501-229 Compilation ended due to lack of space.
1501-224 fatal error in ../exe/xlCcode: signal 9 received.
```

If lack of paging space causes other compiler programs to fail, the following message is displayed:

```
Killed.
```

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

To check the current paging-space settings enter the command: **lsps -a** or use the AIX System Management Interface Tool (SMIT) command **smit pgsp**.

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, 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 Installation Guide. "Setting environment variables" provides a complete list of the required and optional environment variables you can set or reset after installing the compiler, including those used for parallel processing.
- Settings defined in the compiler configuration file, `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 Installation Guide. However, you can customize this file after installation, to specify additional compiler options, default option settings, library search paths, and other settings. Information on customizing the configuration file is provided in "Using custom compiler configuration files" on page 38.
- Settings defined by the GCC options configuration file. If you are using the gxlc utility to map GCC options, the default option mappings are defined in the /opt/IBM/xlc/13.1.3/etc/gxlc.cfg file. You can customize this file to suit your requirements. For more information, see "Configuring the gxlc option mapping" on page 42.

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 and applications you have compiled with it:

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

Compile-time and link-time environment variables

The following environment variables are used by the compiler when you are compiling and linking your code. Many are built into the AIX 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 Installation Guide*. For more information on setting the *LANG* environment variable to use an alternate locale, see your operating system documentation.

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 error messages in the *XL C Installation Guide*.

OBJECT_MODE
Optionally specifies the bit mode for compilation to either 32 or 64 bits. This is equivalent to the **-q32** and **-q64** compiler options. Set the *OBJECT_MODE* environment variable to a value of 32 for 32-bit compilation mode, or 64 for 64-bit compilation mode. If unspecified, the default compilation mode is 32 bits. See also “-q32, -q64” on page 94 for more information.

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.

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

XLC_USR_CONFIG
Specifies the location of a custom configuration file to be used by the compiler. The file name must be given with its absolute path. The compiler will first process the definitions in this file before processing those in the default system configuration file, or those in a customized file specified by the **-F** option; for more information, see “Using custom compiler configuration files” on page 38.

Runtime environment variables

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

LIBPATH
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.

MALLOCALIGN=16
Specifies that dynamic memory allocations return 16-byte aligned addresses. See also "-qipa" on page 193.

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 247 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.

XL_AR
To use your own archive files when generating a nonexecutable package with `-r -qipa=relink`, you can use the **ar** tool and set the *XL_AR* environment variable to point to it. See -qipa for more information.

Environment variables for parallel processing

The XLSMPOPTS environment variable sets options for program run time using loop parallelization. For more information about the suboptions for the **XLSMPOPTS** environment variables, see "XLSMPOPTS" on page 26.

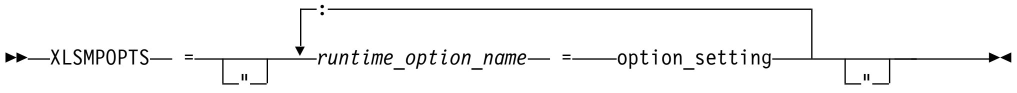
If you are using OpenMP constructs for parallelization, you can also specify runtime options using the OMP environment variables, as discussed in “Environment variables for OpenMP” on page 31.

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

Related information
- “Pragma directives for parallel processing” on page 380
- “Built-in functions for parallel processing” on page 612

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 loops to which no other scheduling algorithm has been explicitly assigned in the source code.

Work is assigned to threads in a different manner, depending on the scheduling type and chunk size used. Choosing chunking granularity is a tradeoff between overhead and load balancing. The syntax for this option is **`schedule`**=*suboption*, where the suboptions are defined as follows:

`affinity[=`*n*`]`

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

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

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

The **affinity** scheduling type 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.

usrthds=*num*

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

stack=*num*

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

Set *num* so it is within the acceptable upper limit. *num* can be up to 256 MB for 32-bit mode, or up to the limit imposed by system resources for 64-bit mode. An application that exceeds the upper limit may cause a segmentation fault.

stackcheck[=*num***]**

When the **-qsmp=stackcheck** is in effect, enables stack overflow checking for slave threads at runtime. *num* is the size of the stack in bytes, 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 281.

startproc=*cpu_id*

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

procs=cpu_id*[,cpu_id,...]*

Enables thread binding and specifies a list of *cpu_id* to which the threads are bound.

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.

bind=*SDL=n1***,***n2***,***n3*

Specifies different system detail levels to bind threads by using the Resource Set API. This suboption supports binding a thread to multiple logical processors.

SDL stands for System Detail Level and can be MCM, L2CACHE, PROC_CORE, or PROC. If the *SDL* value is not specified, or an incorrect *SDL* value is specified, the SMP runtime issues an error message.

The list of three integers *n1,n2,n3* determines how to divide threads among resources (one of SDLs). *n1* is the starting *resource_id*, *n2* is the number of requested resources, and *n3* is the stride, which specifies the increment used to determine the next *resource_id* to bind. *n1,n2,n3* must all be specified; otherwise, the SMP runtime issues an error message and default binding rules apply.

When the number of resources specified in **bind** is greater than the number of threads, the extra resources are ignored.

When the number of threads *t* is greater than the number of resources *x*, *t* threads are divided among *x* resources according to the following formula:

The *ceil(t/x)* threads are bound to the first *(t mod x)* resources. The *floor(t/x)* threads will be bound to the remaining resources.

With the **XLSMPOPTS** environment variable being set as in the following example, a program runs with 16 threads. It binds threads to PROC 0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30.

```
XLSMPOPTS="bind=PROC=0,16,2"
```

Notes:

- The **bind** suboption takes precedence over the **startproc**/**stride** and **procs** suboptions. However, **bindlist** takes precedence over **bind**.
- Resource Set can only be used by a user account with the CAP_NUMA_ATTACH and CAP_PROPAGATE capabilities. These capabilities are set on a per-user basis by using the **chuser** command as follows:

  ```
  chuser "capabilities=CAP_PROPAGATE,CAP_NUMA_ATTACH" username
  ```
- If the *resource_id* specified in **bind** is outside the range of 0 to INT32_MAX, where INT32_MAX is 2147483647 as defined in `stdint.h`, the SMP runtime issues an error message and default binding rules apply.
- The SMP runtime verifies that the *resource_id* exists. If the *resource_id* does not exist, a warning message is issued and the thread is left unbound.
- If you change the number of threads inside the program, for example, through **omp_set_num_threads()** or **num_threads** clause, the following situation occurs:
  - If the number of threads in the application is increased, rebinding takes place based on the environment variable settings.
  - If the number of threads is reduced after binding, the original binding remains.

bindlist=*SDL***=***i1***,***i2***,...***ix*

Specifies different system detail levels to bind threads by using the Resource Set API. This suboption supports binding a thread to multiple logical processors.

SDL stands for System Detail Level and can be MCM, L2CACHE, PROC_CORE, or PROC. If the *SDL* value is not specified, or an incorrect *SDL* value is specified, the SMP runtime issues an error message.

The list of *x* integers *i1,i2...ix* enumerates the resources (one of SDLs) to be used during binding. When the number of integers in the list is greater than or equal to the number of threads, the position in the list determines the thread ID that will be bound to the resource.

When the number of resources specified in **bindlist** is greater than the number of threads, the extra resources are ignored.

When the number of threads *t* is greater than the number of resources *x*, *t* threads will be divided among *x* resources according to the following formula:

The *ceil(t/x)* threads are bound to the first *(t mod x)* resources. The *floor(t/x)* threads will be bound to the remaining resources.

For example:

```
XLSMPOPTS="bindlist=MCM=0,1,2,3"
```

This example code shows that threads are bound to MCM 0,1,2,3. When the program runs with four threads, thread 0 is bound to MCM 0, thread 1 is bound to MCM 1, thread 2 is bound to MCM 2, and thread 3 is bound to

MCM 3. When the program runs with six threads, threads 0 and 1 are bound to MCM 0, threads 2 and 3 are bound to MCM 1, thread 4 is bound to MCM 2, and thread 5 is bound to MCM 3.

With the **XLSMPOPTS** environment variable being set as in the following example, a program runs with eight (or fewer) threads. It binds all even-numbered threads to L2CACHE 0 and all odd-numbered threads to L2CACHE 1.

```
XLSMPOPTS="bindlist=L2CACHE=0,1,0,1,0,1,0,1"
```

Notes:

- The **bindlist** suboption takes precedence over the **startproc**/**stride**, **procs**, and **bind** suboptions.
- Resource Set can only be used by a user account with the CAP_NUMA_ATTACH and CAP_PROPAGATE capabilities. These capabilities are set on a per-user basis by using the **chuser** command as follows:

  ```
  chuser "capabilities=CAP_PROPAGATE,CAP_NUMA_ATTACH" username
  ```
- The SMP runtime verifies that the thread ID specified for a resource is not less than 0 nor greater than the available resources. Otherwise, the SMP runtime issues a warning message and the thread is left unbound.
- If you change the number of threads inside the program, for example, through **omp_set_num_threads()** or **num_threads** clause, the following situation occurs:
  - If the number of threads in the application is increased, rebinding takes place based on the environment variable settings.
  - If the number of threads is reduced after binding, the original binding remains.

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**.

Environment variables for OpenMP

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

```
►►—env_variable—=—option_and_args————————————————————————————►◄
```

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

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

OMP_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:

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

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

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

FALSE
: Instructs the runtime environment not to display any information.

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

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 26

"OMP_PROC_BIND" on page 35

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

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

If **OMP_DYNAMIC** is set to **TRUE**, dynamic adjustment is enabled. The number of threads that are available for executing parallel regions can be adjusted at run time to make the best use of system resources. For more information, see the description for **profilefreq=***num* in "XLSMPOPTS" on page 26.

If **OMP_DYNAMIC** is set to **FALSE**, dynamic adjustment is disabled.

The default setting is TRUE.

Related information

"OMP_PROC_BIND" on page 35

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

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

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

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

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

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

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

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

The setting of the **omp_set_nested** routine overrides the **OMP_NESTED** setting. The **OMP_NESTED** setting overrides the setting of the **-qsmp=nested_par | nonested_par** option.

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

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

The syntax of the environment variable is as follows:

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

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

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

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

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

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

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

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

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

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

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

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

Example

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

// omp_get_max_threads() returns 3

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

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

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

OMP_PROC_BIND: The **OMP_PROC_BIND** environment variable controls whether OpenMP threads can be moved between places.

OMP_PROC_BIND syntax

```
►►──OMP_PROC_BIND=──┬─TRUE──┬──────────────────────────────────────────────►◄
                    └─FALSE─┘
```

TRUE
: Binds the threads to places.

FALSE
: Allows threads to be moved between places.

Usage

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

Table 8. Thread binding rule summary

OMP_PROC_BIND settings	XLSMPOPTS settings	Thread binding results
OMP_PROC_BIND is not set	**XLSMPOPTS** is not set.	Threads are not bound.
	XLSMPOPTS is set to **startproc**/**stride**, **procs**, **bind**, or **bindlist**.	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**, **procs**, **bind**, or **bindlist**.	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**, **procs**, **bind**, or **bindlist**.	
	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.

The **OMP_PROC_BIND** environment variable provides a portable way to control whether OpenMP threads can be migrated. The **startproc**/**stride**, **procs**, **bind**, or **bindlist** 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.

Related information:

"XLSMPOPTS" on page 26

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 622

"omp_get_schedule" on page 622

OMP_STACKSIZE:

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

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

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

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

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

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

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

If the value of **OMP_STACKSIZE** is not set, the initial value is set to the default value. The default value is 4194304B. The maximum value for 32-bit mode is 256M. For 64-bit mode, the maximum is up to the limit imposed by system resources.

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

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

OMP_THREAD_LIMIT:

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

```
►►──OMP_THREAD_LIMIT──=──n─────────────────────────────────────────────────►◄
```

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

Usage

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

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

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

If neither the **OMP_THREAD_LIMIT** nor **OMP_NUM_THREADS** environment variable is set, the default value for **OMP_THREAD_LIMIT** is the number of available processors.

Related information:

"OMP_NUM_THREADS" on page 34

OMP_WAIT_POLICY:

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

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

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

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

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

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

Using custom compiler configuration files

The XL C compiler generates a default configuration file /opt/IBM/xlc/13.1.3/etc/xlc.cfg.*nn* , where *nn* indicates which OS version the configuration file is for). 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 143

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 9. Assignment operators and attribute ordering

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

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

Custom user-defined configuration file ~/userconfig2

```
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

xlc_r: use=xlc_r
    options+=-qlist
```

This example specifies that **-qlist** is to be used for any compilation called by the **xlc** and **xlc_r** commands. 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.

Configuring the gxlc option mapping

The **gxlc** utility uses the configuration file /opt/IBM/xlc/13.1.3/etc/gxlc.cfg to translate GNU C options to corresponding XL C options. Each entry in gxlc.cfg describes how the utility should map a GNU C option to an XL C option and how to process it.

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

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

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

```
abcd    "gcc_option"    "xlc_option"
```

where:

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

```
ynn*       "-finline"               "-qinline"
```

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

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

```
nyn*       "-fmyvalue"          "-qmyvalue"
```

The utility will then expect a value for these options.

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

n
Tells the utility to process the option listed in the *gcc_option* field.

i
Tells the utility to ignore the option listed in the *gcc_option* field. The utility will generate a message that this has been done, and continue processing the given options.

e
Tells the utility to halt processing if the option listed in the *gcc_option* field is encountered. The utility will also generate an error message.

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

```
nni*       "-I-"
```

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

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

c
Tells the utility to translate the option only for C.

* Tells **gxlc** or **gxlc++** to translate the option for C.

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

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

"*gcc_option*"
: Is a string representing a GNU C option. This field is required and must appear in double quotation marks.

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

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

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

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

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

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

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

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

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

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

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

For a complete list of GNU C and XL C option mappings, see the following web page: http://www.ibm.com/support/docview.wss?uid=swg27039014

Related information

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

Chapter 3. Tracking and reporting compiler usage

You can use the utilization tracking and reporting feature to record and analyze which users in your organization are using the compiler and the number of users using it concurrently. This information can help you determine whether your organization's use of the compiler exceeds your compiler license entitlements.

To use this feature, follow these steps:

1. Understand how the feature works. See "Understanding utilization tracking and reporting" for more information.
2. Investigate how the compiler is used in your organization, and decide how you track the compiler usage accordingly. See "Preparing to use this feature" on page 54 for more information.
3. Configure and enable utilization tracking. See "Configuring utilization tracking" on page 60 for more information.
4. Use the utilization reporting tool to generate usage reports or prune usage files. See "Generating usage reports" on page 68 or "Pruning usage files" on page 71 for more information.

Understanding utilization tracking and reporting

The utilization tracking and reporting feature provides a mechanism for you to detect whether your organization's use of the compiler exceeds your compiler license entitlements. This section introduces the feature, describes how it works, and illustrates its typical usage scenarios.

Overview

When utilization tracking is enabled, all compiler invocations are recorded in a file. This file is called a usage file and it has the `.cuf` suffix. You can then use the utilization reporting tool to generate a report from one or more of these usage files, and optionally prune the usage files.

You can use the utilization tracking and reporting feature in various ways based on how the compiler is used in your organization. The "Four usage scenarios" on page 46 section illustrates the typical usage scenarios of this feature.

The following sections introduce the configuration of the utilization tracking functionality and the usage of the utilization reporting tool.

Utilization tracking

A utilization tracking configuration file `urtxlc1302aix.cfg` is included in the default compiler installation. You can use this file to enable utilization tracking and control different aspects of the tracking.

A symlink `urt_client.cfg` is also included in the default compiler installation. It points to the location of the utilization tracking configuration file. If you want to put the utilization tracking configuration file in a different location, you can modify the symlink accordingly.

For more information, see "Configuring utilization tracking" on page 60.

Note: Utilization tracking is disabled by default.

Utilization reporting tool

The utilization reporting tool generates compiler usage reports based on the information in the usage files. You can optionally prune the usage files with the tool. For more information, see "Generating usage reports" on page 68 and "Pruning usage files" on page 71.

Four usage scenarios

This section describes four possible scenarios for managing the compiler usage, for recording the compiler usage information and for generating reports from this information.

The following scenarios describe some typical ways that your organization might be using the compiler and illustrate how you can use this feature to track compiler usage in each case.

Note: Actual usage is not limited to these scenarios.

"Scenario: One machine, one shared .cuf file"

"Scenario: One machine, multiple .cuf files" on page 47

"Scenario: Multiple machines, one shared .cuf file" on page 50

"Scenario: Multiple machines, multiple .cuf files" on page 52

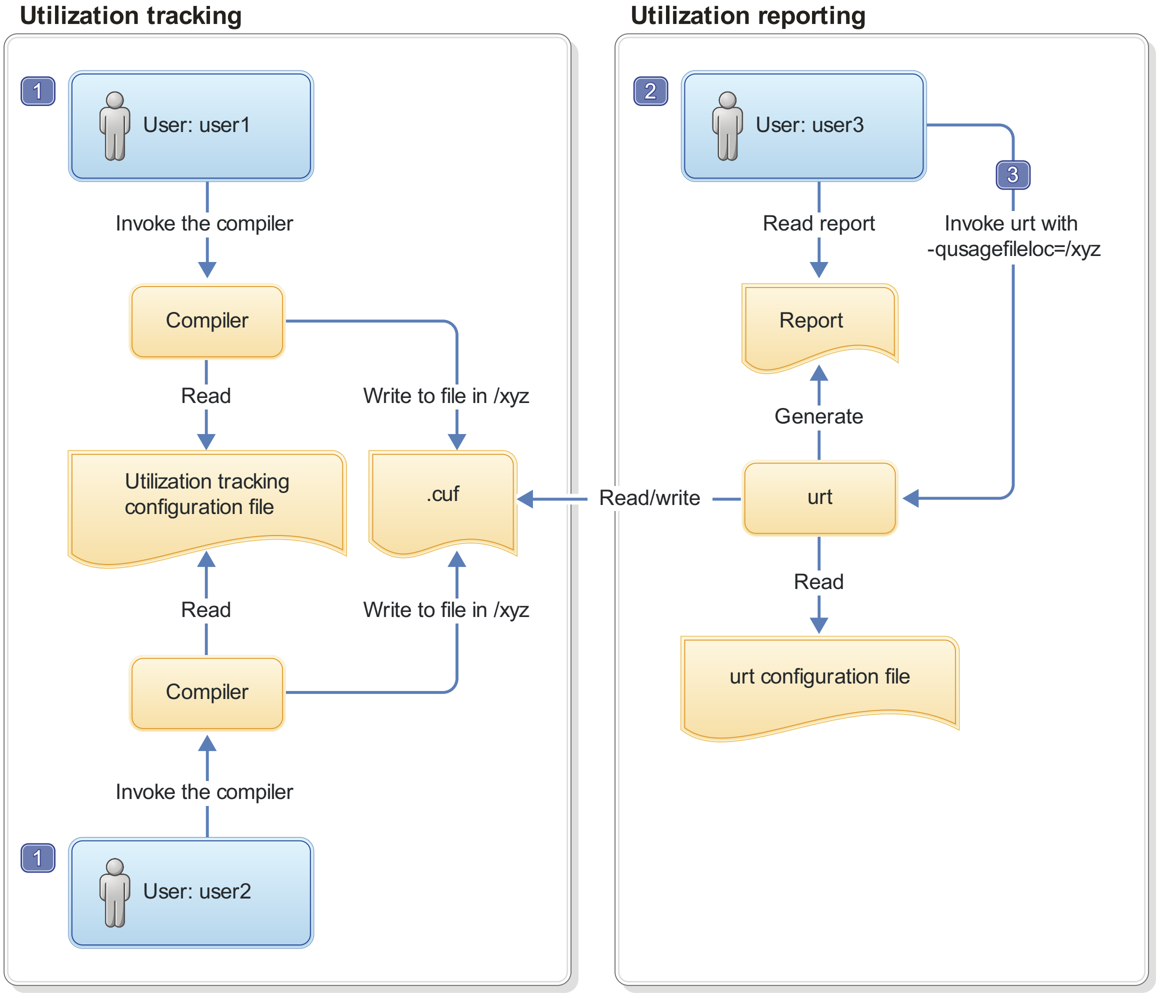
Scenario: One machine, one shared .cuf file

This scenario describes an environment where all the compilations are done on one machine and all users share one `.cuf` file.

The advantage of using the approach in this scenario is that it simplifies report generation and usage file pruning, because the utilization report tool only need to access one `.cuf` file. The disadvantage is that all compiler users need to compete for access to this file. Because the file might become large, it might have an impact on performance. Some setup work is also required to create the shared `.cuf` file and to give all compiler users write access. The "The number of usage files" on page 57 section provides detailed information about using a single usage file for all compiler users.

In this scenario, compiler users run the compiler on the same machine and their utilization information is recorded in a shared `.cuf` file. The utilization tracking configuration file for the compiler is modified to point to the location of the `.cuf` file. When the compiler is invoked, it writes the utilization information to that file. You can then use the utilization reporting tool to retrieve the utilization information from the file and generate usage reports.

The following diagram illustrates this scenario.



1. Both user1 and user2 need write access to the `.cuf` file in `/xyz`.
2. user3 needs read access to the `.cuf` file in`/xyz` to generate the usage report, and write access to prune the `.cuf` file.
3. A cron job can be created to run **urt** automatically on a regular basis.

Figure 5. Compiler users use a single machine, with a shared .cuf file

The diagram reflects the following points:

1. user1 and user2 use the same utilization tracking configuration file, which manages the tracking functionality centrally. A common location `/xyz` is created to keep a shared `.cuf` file.
2. When user1 and user2 invoke the compiler, the utilization information is recorded in the `.cuf` file under the common directory `/xyz`.
3. user3 invokes **urt** with `-qusagefileloc=/xyz` to generate usage reports.

Note: Regular running of the utilization reporting tool can prevent these files from growing too big, because you can prune the usage files with this tool.

Scenario: One machine, multiple .cuf files

This scenario describes an environment where all the compilations are done on one machine and all users have their own `.cuf` files.

The approach in this scenario has the following advantages:

- Compiler users do not have to compete for access to a single `.cuf` file, and this might result in better performance.
- You do not need to set up write access to a single common location for all compiler users. They already have write access to their own home directories.

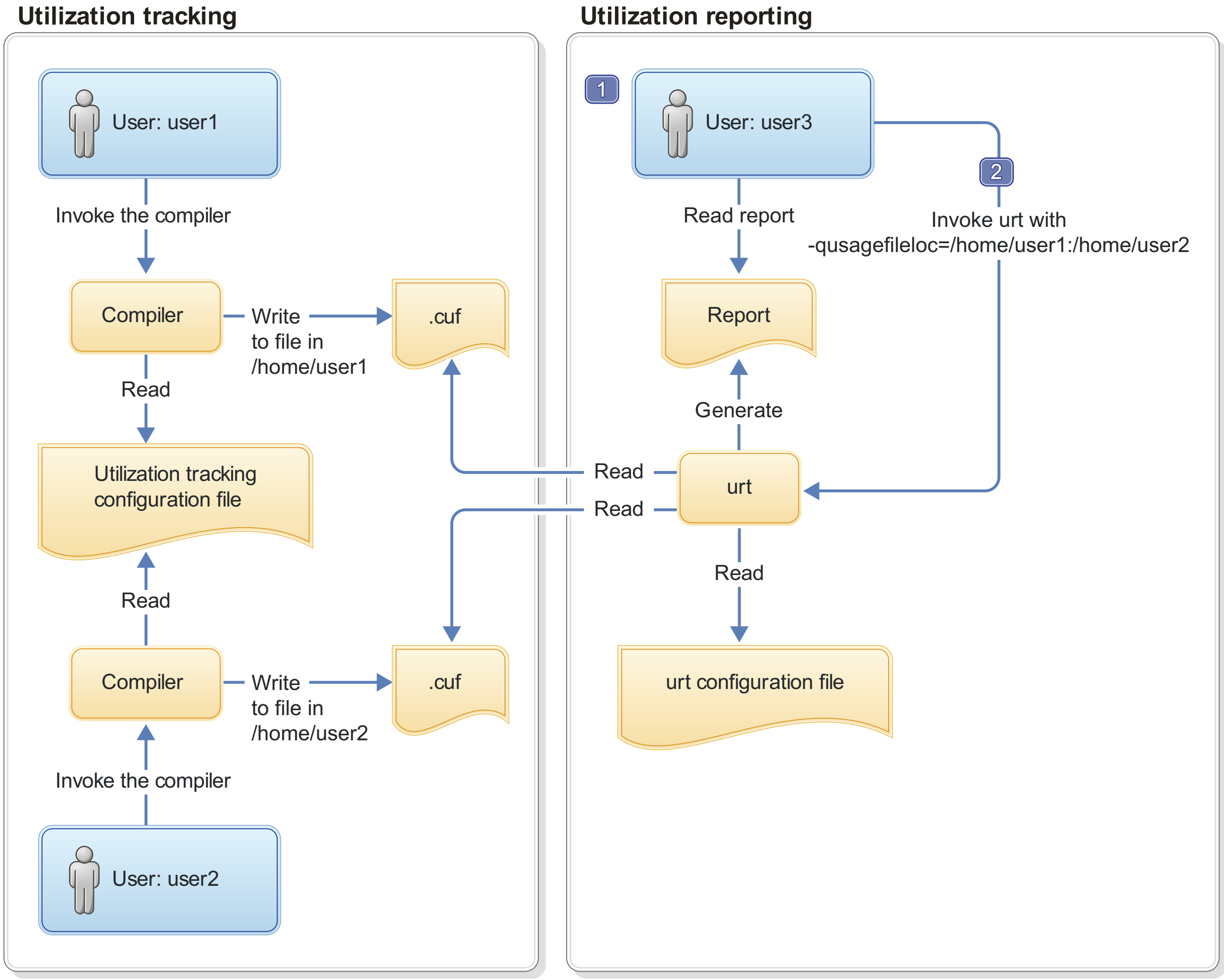
However, using multiple `.cuf` files that are automatically created in each user's home directory might have the following issues:

- Compiler users might not know that the file has been created or what it is when they see the file. In this case, they might delete the file.
- Some users' home directories might be on file systems that are mounted from a remote system. This causes utilization tracking to use a remote file, which might affect performance.
- Compiler users might not want `.cuf` files to take up space in their home directories.

Instead of using each user's home directory, the `.cuf` files for each user can be created in a common location. The "Usage file location" on page 56 section provides detailed information about how to create these files in a common location.

In this scenario, two compiler users run the compiler on the same machine and they have their own `.cuf` files. When the compiler is invoked, it automatically creates a `.cuf` file for each user and writes the utilization information to that file. You can then use the utilization reporting tool to retrieve the utilization information from the `.cuf` files and generate usage reports.

The following diagram illustrates this scenario.



1. user3 needs read access to `.cuf` files in `/home/user1` and `/home/user2` to generate the usage report, and write access to prune the usage files.
2. A cron job can be created to run **urt** automatically on a regular basis.

Figure 6. Compiler users use one machine, with separate .cuf files

This diagram reflects the following points:

1. user1 and user2 use the same utilization tracking configuration file, which manages the tracking functionality centrally.
2. When user1 and user2 invoke the compiler, the utilization information is recorded in the two `.cuf` files under their respective home directories, `/home/user1` and `/home/user2`.
3. user3 invokes **urt** with `-qusagefileloc=/home/user1:/home/user2` to generate usage reports.

   Note: If you need to find out which home directories contain usage files, you can invoke **urt** as follows:

   ```
   urt -qusagefileloc=/home -qmaxsubdirs=1
   ```

   In this case, **urt** looks for all the `.cuf` files under `/home` directory.

Scenario: Multiple machines, one shared .cuf file

This scenario describes an environment where the compilations are done on multiple machines but all users share a single `.cuf` file.

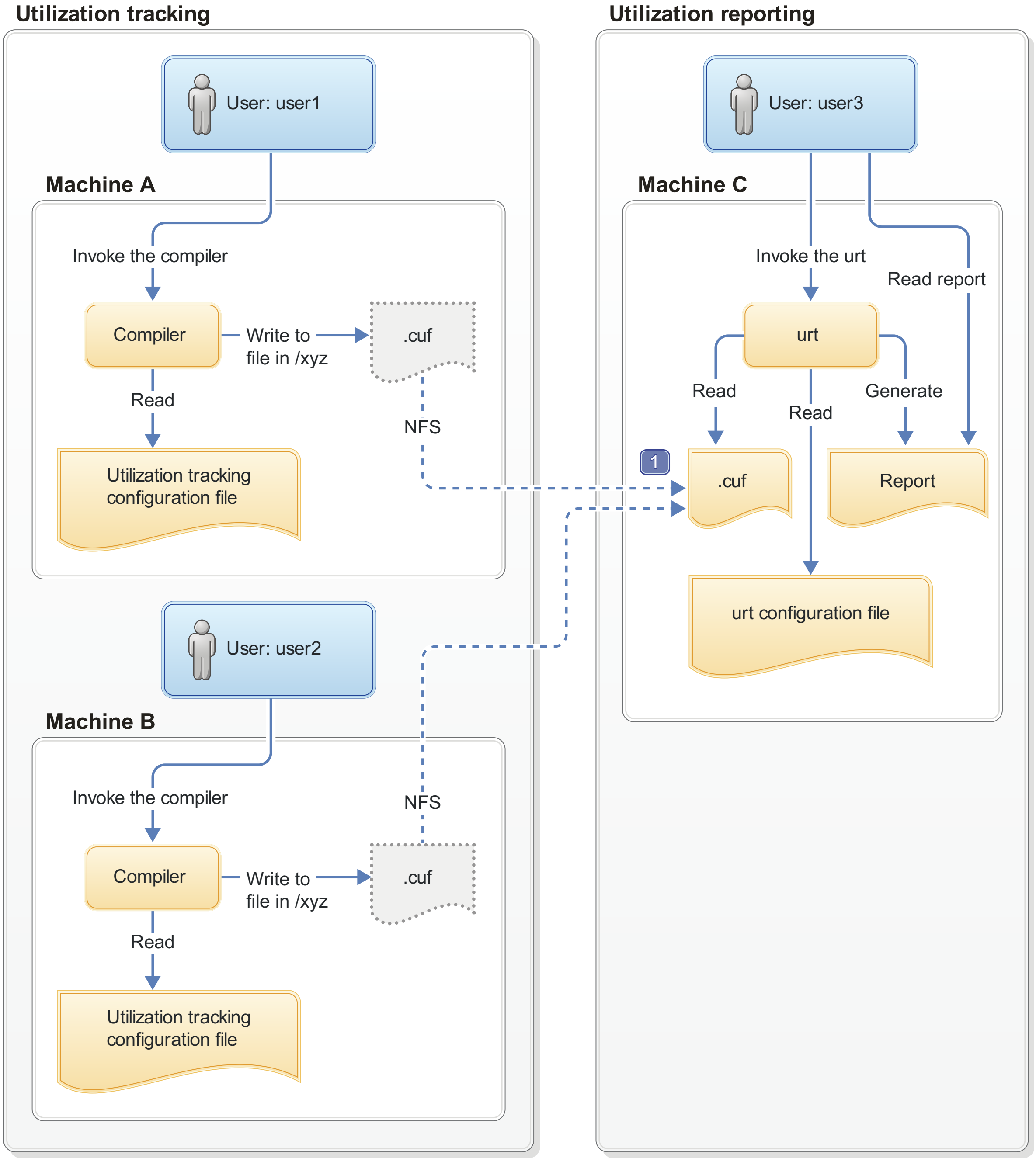
The advantage of the approach in this scenario is that using one `.cuf` file can simplify the report generation and the usage file pruning process. The section "The number of usage files" on page 57 provides detailed information about using a single usage file for all compiler users. The `.cuf` file is already on the machine where the utilization reporting tool is installed. You do not need to copy the file to that machine or install the tool on multiple machines to prune the `.cuf` files.

This approach has the following disadvantages:

- The compiler users must compete for access to one usage file. Because the file might become large, it might have an impact on performance.
- Some setup work is required to create the shared `.cuf` file and to give all compiler users write access on a network file system.
- The efficiency of the whole process depends on the speed and reliability of the network file system, because the compilers and the `.cuf` file are on different machines. For example, some file systems are better than others in supporting file locking, which is required for concurrent access by multiple users.

In this scenario, two compiler users run the compilers on separate machines and they use one shared `.cuf` file on a network file system, such as NFS, DFS, or AFS™. When the compiler is invoked, it writes the utilization information to that file. You can then use the utilization reporting tool to retrieve the utilization information from the file and generate usage reports.

The following diagram illustrates this scenario.



1. On Machine A and Machine B, mount point /xyz is created to Machine C. All compiler utilization is recorded in the .cuf file, from which the usage report is generated.

Figure 7. Compiler users use multiple machines, with a shared .cuf file

This diagram reflects the following points:

1. Utilization tracking is configured respectively on Machine A and Machine B.

   Notes:

   - Although each machine has its own configuration file, the contents of these files must be the same.
   - Centrally managing the utilization tracking functionality can reduce your configuration effort and eliminate possible errors. The "Central

configuration" on page 55 section provides detailed information about how you can use a common configuration file shared by compiler users using different machines.

2. A network file system is set up for the central management of the `.cuf` files. When user1 and user2 invoke the compilers from Machine A and Machine B, the utilization information of both compilers is written to the `.cuf` file on Machine C.
3. user3 invokes **urt** to generate usage reports from the `.cuf` file on Machine C.

Note: You can use the utilization reporting tool to prune the usage files regularly to prevent them from growing too big.

Scenario: Multiple machines, multiple .cuf files

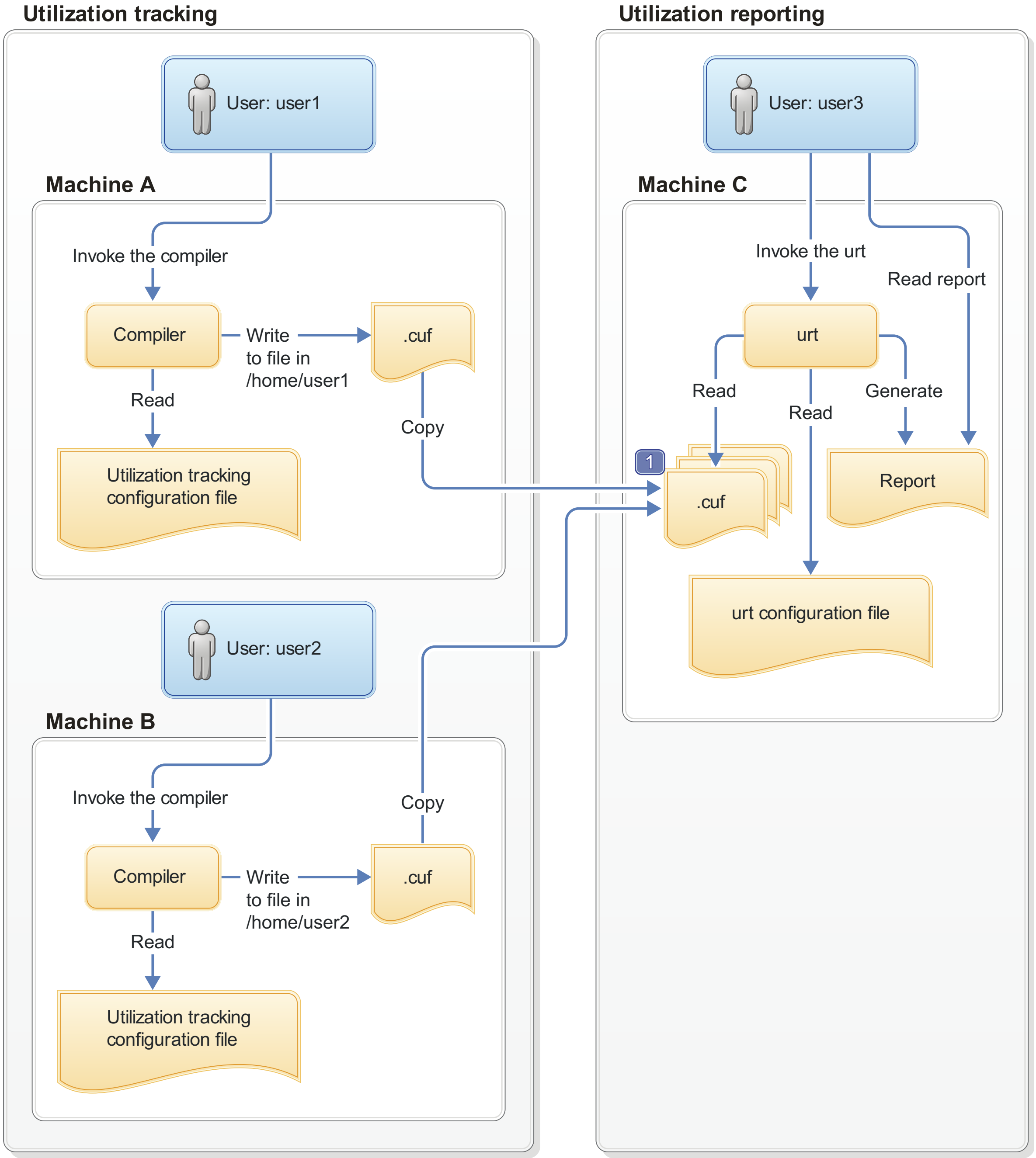
This scenario describes an environment where the compilations are done on multiple machines and all users have their own usage files.

In this scenario, two compiler users run the compilers on separate machines and they have their own `.cuf` files. When the compiler is invoked, it writes the utilization information to that file. You can then use the utilization reporting tool to retrieve the utilization information from the file and generate usage reports. This tool can be run on either of the machines on which the compiler is installed or on a different machine.

Note: The utilization reporting tool requires read access to all the `.cuf` files. You can use either of the following methods to make the files accessible in this example:

- Use a network file system, such as NFS, DFS, or AFS.
- Copy the files from their original locations to the machine where you plan to run the utilization reporting tool. You can use **ftp**, **rcp**, **rsync** or any other remote copy command to copy the files.

The following diagram illustrates this scenario.



1. user3 copies the `.cuf` files to Machine C. A cron job can be created to copy the files automatically on a regular basis.

Figure 8. Compiler users use multiple machines, with multiple .cuf files

This diagram reflects the following points:

1. Utilization tracking is configured respectively on Machine A and Machine B.

   Notes:

   - Although each machine has its own configuration file, the contents of these files must be the same.
   - Centrally managing the utilization tracking functionality can reduce your configuration effort and eliminate possible errors. The "Central

configuration" on page 55 section provides detailed information about how you can use a common configuration file shared by compiler users using different machines.

2. When user1 and user2 invoke the compilers, the utilization information is recorded in the two `.cuf` files under their respective home directories, `/home/user1` and `/home/user2`.

   Note: These `.cuf` files can also be created in another common location, for example, `/var/tmp`. The "Usage file location" on page 56 section provides detailed information about how to create these files in a common location.
3. user3 copies the two `.cuf` files from Machine A and Machine B to Machine C.
4. user3 invokes **urt** to generate usage reports from the `.cuf` files on Machine C.

Related information

- "Preparing to use this feature"
- "Configuring utilization tracking" on page 60
- "Generating usage reports" on page 68
- "Pruning usage files" on page 71

Preparing to use this feature

Before enabling utilization tracking within your organization, you must consider certain factors related to how the compiler is used in your organization.

The following sections describe those considerations in detail:

Time synchronization

If you plan to track the utilization of the compiler on more than one machine, you must consider synchronizing the time across the machines.

The usage report generated by the utilization reporting tool lists the time when the compiler invocations start and end. The report also determines which invocations are concurrent. The accuracy and validity of this information will be affected if time is not synchronized across these machines.

If you are unable to synchronize time across different machines, you can use the **-qadjusttime** option to instruct the utilization reporting tool to adjust the times that have been recorded.

License types and user information

Before you start to use this feature, you need the number and type of license entitlements for your organization.

The license and user information that you need are as follows:

- The number of Concurrent User licenses that you have for this compiler. This information is required for the **-qmaxconcurrentusers** entry in the utilization tracking configuration file.
- The users who have their own Authorized User license for this compiler. This information is used for the **-qexemptconcurrentusers** entry in the utilization tracking configuration file.
- The users who use the compiler with multiple accounts. This information is used for the **-qsameuser** option for the utilization reporting tool.

Note: It is not mandatory to specify the users who have their own Authorized User license and the users who use the compiler with multiple accounts, but specifying them improves the accuracy of the usage reports generated by the utilization reporting tool. For detailed information, see "Concurrent user considerations."

Central configuration

Configuring utilization tracking the same for all compiler users is very important, because it can ensure the accuracy of your utilization tracking, and minimize your configuration and maintenance effort. You can achieve this by ensuring that all users use the same utilization tracking configuration file.

If you have only one installation of the compiler, you can directly edit the utilization tracking configuration file. Every compiler user can automatically use that configuration file.

If you have multiple installations of the compiler, you need to maintain a single utilization tracking config file and reference it from all installations. Any changes you make to the utilization tracking configuration file, including enabling or disabling utilization tracking, can automatically apply to all compiler installations when users invoke the compiler. In each installation, there is a symlink named `urt_client.cfg`, located in `/opt/IBM/xlc/13.1.3/urt`. Modify the symlink to point to this shared instance of the configuration file.

If the compiler is installed on multiple machines, the utilization tracking configuration file needs to be placed on a network file system, such as NFS, DFS, or AFS, to be used by the compiler on each machine.

Note: If it is not possible for you to use a single utilization tracking configuration file for all compiler users, you must ensure all utilization tracking configuration files for each compiler installation are consistent. Using different configurations for the same compiler is not supported.

Concurrent user considerations

Invocations of the compiler are considered concurrent when their start time and end times overlap. This section provides the information about how the utilization reporting tool counts concurrent users and the ways to increase the accuracy of the usage reports.

When the utilization reporting tool counts concurrent users, it looks at the user account information that has been captured in the usage files. The account information consists of a user name, a user ID, and a host name. By default, each unique combination of this account information is considered and counted as a different user. However, invocations of the compiler by the following users must not be included in the count of concurrent users:

- Users who have their own Authorized User license are considered exempt users, because their use of the compiler does not consume any Concurrent User licences.
- Users who have multiple accounts. Because the accounts belong to the same user, invocations of the compiler while logged on using those accounts are counted as usage by a single user.

The utilization reporting tool can account for the above situations if you provide it with information regarding exempt users and users with multiple accounts. Here is how you can provide the information:

- Specify the **-qexemptconcurrentusers** entry in the utilization tracking configuration file. This entry specifies users with Authorized User licenses.
- Specify the **-qsameuser urt** command-line option. This option specifies users with multiple accounts.

Notes:

- When the number of concurrent users is adjusted with **-qexemptconcurrentusers** or **-qsameuser**, the utilization reporting tool generates a message to indicate that the concurrent usage information is adjusted.
- The number of concurrent users might be zero if all concurrent invocations are invoked by exempt users. The tool also generates a message with this information.

Usage file considerations

Usage (`.cuf`) files are used to store compiler usage information. This section provides information that helps you decide how you want to generate and use these files.

Usage file location

Usage files can be created in each user's home directory, or they can be created in a central location for all users.

With utilization tracking enabled, when a compiler user compiles a program, a `.cuf` file is automatically created in the user's home directory in case the file does not exist. This is convenient for testing the utilization tracking feature because users already have write access to their own home directories, which means no additional setup is required. However, this might have the following issues:

- Compiler users might not know that the file has been created or what it is when they see the file. In this case, they might delete the file.
- Some users' home directories might be on file systems that are mounted from a remote system. This causes utilization tracking to use a remote file, which might affect performance.
- Compiler users might not want usage files to take up space in their home directory.

A good alternative is to set up a central location where the usage files can be created, and provide appropriate access to that location for both the compiler users and the utilization reporting tool users. This can be set up by using the `other/world` permissions or by using group permissions.

For example, if the central location is a directory named `/var/tmp/track_compiler_use`, you can modify the **-qusagefileloc** entry in the utilization tracking configuration file as follows:

```
-qusagefileloc=/var/tmp/track_compiler_use/$LOGNAME.cuf
```

This creates a `.cuf` file for each user in the specified location, such as `user1.cuf` or `user2.cuf`. It is easier to run the utilization reporting tool to generate the usage report from the `.cuf` files in this central location. You only need to pass the path of the location, `/var/tmp/track_compiler_use` to the utilization reporting tool , and then the tool can read all the `.cuf` files in that location.

If the compiler users are running the compiler on more than one machine, you need to add *$HOSTNAME* to the **-qusagefileloc** entry to ensure that there are no collisions in the file names. For example, you can specify the **-qusagefileloc** entry as follows:

```
-qusagefileloc=/var/tmp/track_compiler_use/$HOSTNAME_$LOGNAME.cuf
```

This creates a `.cuf` file for each user, and the name of that `.cuf` file also contains the name of the host on which the compiler is used, such as `host1_user1.cuf`.

The number of usage files

You can use one usage file or separate usage files for different compiler users.

Using separate usage files for different compiler users

The advantages of using separate usage files are as follows:

- It might provide better performance because compiler users access their own usage files instead of competing for access to a shared one and separate usage files are usually smaller.
- Usage file for a user can be automatically created when the user uses the compiler to compile a program. There is no need to explicitly create a usage file for each user beforehand. For more information, see "Usage file location" on page 56.
- When generating utilization reports, you usually include all compiler users. However, if there are circumstances in which you want to exclude some users, you can simply omit their usage files when you invoke the utilization reporting tool. For example, you might want to omit users who have their own Authorized User license.

The disadvantage is that you might have to maintain separate usage files for different users.

Using a single usage file for all compiler users

The advantage of using a shared usage file for all users is that you only need to maintain a single file instead of multiple files. However, with a single usage file, you lose the flexibility and possible performance benefits of using multiple usage files, as described in the preceding subsection.

The compiler provides an empty usage file `urtstub.cuf` in the `opt/IBM/xlc/13.1.3/urt` directory. You can create a usage file for all compiler users by copying the empty usage file to a directory where they all have write access. In this case, you need to change the **-qusagefileloc** entry in the utilization tracking configuration file to point to the location of the usage file.

Usage files on multiple machines

If you use the compiler on multiple machines, you need to decide how to make the usage files available for the utilization reporting tool.

You can use various methods to make the usage files available for the utilization reporting tool to generate usage reports and prune the usage files. Choose one of the following approaches to manage usage files on multiple machines:

- Copy the usage files from the machines where the compiler is used to the machine where the utilization reporting tool is installed. You can use any remote copy command, for example, **ftp**, **rcp**, **scp**, and **rsync**. In this case, the usage files are being accessed locally by both the compiler, for utilization tracking, and by the utilization reporting tool, for generating the usage report. Accessing the files locally yields the best performance.

- Use a distributed file system to export the file system from the machines where the compiler is used, and mount those file systems on the machine where the utilization reporting tool is installed. When you run the utilization reporting tool, it can access the usage files remotely via the mounted file systems.
- You can also export the file system from the machine where the utilization reporting tool is installed, and mount that file system on each machine where the compiler is used, using it as the location of the usage files where the compiler is recording its utilization. In this approach, the compiler records utilization in a remote usage file, and the utilization reporting tool reads the usage file locally.

  Note: If you find this degrades the performance of the compiler, consider using one of the first two approaches instead.

Usage file size

You need to consider the fact that the size of the usage files might grow quickly, especially when you use a shared file for all compiler users. If the usage file gets too large, it might affect utilization tracking performance.

To keep the usage files from growing quickly, you can optionally prune the usage files when you generate usage reports. You can also prune the files regularly using cron.

For more information about how to prune files, see "Pruning usage files" on page 71.

Regular utilization checking

You can run the utilization reporting tool on a regular basis to verify whether the usage of the compiler is compliant with the Concurrent User licenses you have purchased. You can create a cron job to do this automatically.

If the usage files need to be copied to the machine where the utilization reporting tool is running, you can also automate the copying task with a cron job.

Another reason for running the tool regularly is to prune the usage files to control the size of these files.

Note: To reduce contention for read and write access to the usage files, run the utilization reporting tool or copy the usage files when the compiler is not being used.

Testing utilization tracking

Before you begin to track the compiler usage for all users in your organization, you can test the feature with a limited number of users or with a separate compiler installation. During this testing, you can try different configurations so as to decide the best setup for your organization.

Testing with a limited number of users

To enable compiler utilization tracking for a limited number of users, you can use a separate utilization tracking configuration file and ask only these users to use the file. Other users of the same installation use the default utilization tracking configuration file in which utilization tracking is disabled, and their use of the compiler is therefore not recorded.

The default compiler configuration file, xlc.cfg.61 or xlc.cfg.71 contains two entries, *xlurt_cfg_path* and *xlurt_cfg_name*, which specify the location of the utilization tracking configuration file. You need to perform the following tasks to let the specified users use the separate utilization tracking configuration file:

1. Create a separate compiler configuration file or stanza, in which the *xlurt_cfg_path* and *xlurt_cfg_name* entries specify the location of the utilization tracking configuration file you want to use.
2. Ask these users to use the following compiler option or environment variable to instruct the compiler to use the separate compiler configuration file or stanza, which in turn allows them to use the separate utilization tracking configuration file.
   - The **-F** option
   - The XLC_USR_CONFIG environment variable

Example 1

When you use the default configuration file and a new stanza *xlc_urt* to compile your program `myprogram.c`, follow two steps:

1. Create the stanza in corresponding `xlc.cfg.61` or `xlc.cfg.71` compiler configuration file. For example:

```
xlc_urt: use       =  DEFLT
         xlurt_cfg_path=$location_of_separate_utilization_conf_file
         xlurt_cfg_name=$name_of_separate_utilization_conf_file
         crt       = /lib/crt0.o
         mcrt      = /lib/mcrt0.o
         gcrt      = /lib/gcrt0.o
         libraries  = -L/opt/IBM/xlc/13.1.3/lib,-lxlopt,-lxl,-lc
         proflibs   = -L/lib/profiled,-L/usr/lib/profiled
         options    = -qlanglvl=extc99,-qcpluscmt,-qkeyword=inline,-qalias=ansi
```

2. Use the following command to compile `myprogram.c`:

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

Example 2

When you use the newly created compiler configuration file `myconfig.cfg` to compile your program `myprogram.c`, follow two steps:

1. Set *xlurt_cfg_path* and *xlurt_cfg_name* entries to the location and name of separate utilization tracking configuration file accordingly. For example:

```
DEFLT_C:
        use           =DEFLT
        xlurt_cfg_path=$location_of_separate_utilization_conf_file
        xlurt_cfg_name=$name_of_separate_utilization_conf_file

DEFLT_CPP:
        use           =DEFLT
        xlurt_cfg_path=$location_of_separate_utilization_conf_file
        xlurt_cfg_name=$name_of_separate_utilization_conf_file
```

2. Use either one of the following commands to compile `myprogram.c`:

```
export XLC_USR_CONFIG="$location_of_newly_created_configuration_file/myconfig.cfg"
xlc myprogram.c
```

or

```
xlc myprogram.c -F$location_of_newly_created_configuration_file/myconfig.cfg
```

Note: This approach is only for testing the utilization tracking feature. Do not use it for tracking all compiler utilization in your organization unless you can ensure that all compiler invocations are done with the **-F** option or the **XLC_USR_CONFIG** environment variable set.

Testing with a separate compiler installation

You can install a separate instance of the compiler for testing utilization tracking. In this case, you can directly modify the utilization tracking configuration file in that installation to enable and configure utilization tracking. The compiler users involved in the testing do not need to perform any task for the tracking.

When you are satisfied that you have found the best utilization tracking configuration for your organization, you can enable it for all compiler users in your organization.

Related information

- "Configuring utilization tracking"
- -F

Configuring utilization tracking

You can use the utilization tracking configuration file to enable and configure the utilization tracking functionality.

The default location of the configuration file is `/opt/IBM/xlc/13.1.3/urt` and its file name is `urtxlc1302aix.cfg`.

The compiler uses a symlink to specify the location of the utilization tracking configuration file. The symlink is also located in `/opt/IBM/xlc/13.1.3/urt` and its name is `urt_client.cfg`. In the following situations, you might need to change the symlink:

- If you want to use a utilization tracking configuration file in a different location, change the symlink to point to that location.
- If you have multiple installations of the same compiler, and you plan to use a single utilization tracking configuration file, change the symlink in each installation to point to that file. For more information, see "Central configuration" on page 55.

Note: Installing a PTF update does not overwrite the utilization tracking configuration file.

You can use the entries in the utilization tracking configuration file to configure how compiler usage is tracked. For details about the specific entries in that file and how they can be modified, see "Editing utilization tracking configuration file entries."

Editing utilization tracking configuration file entries

You can configure different aspects of utilization tracking by editing the entries in the utilization tracking configuration file.

The entries are divided into two categories.

1. The entries in the *Product information* category identify the compiler. Do not modify these entries.

2. The entries in the *Tracking configuration* category can be used to configure utilization tracking for this product. Changes to these entries take effect in the usage file upon the next compiler invocation. In this case, the compiler emits a message to indicate that the new configuration values have been saved in the usage file. When you generate a report from the usage file, the new values are used.

The following rules apply when you modify the entries:

- The following entries are written to the usage files whenever you change them, and they are used the next time the utilization reporting tool generates a report from the usage files. These configuration entries must be the same for all compiler users.
  - **-qmaxconcurrentusers**
  - **-qexemptconcurrentusers**
  - **-qqualhostname**
- If **-qqualhostname** is changed, you must discard any existing usage files and start tracking utilization again with new usage files. Otherwise some invocations are recorded with qualified host names and some are recorded with unqualified host names.

Notes:

- The entries are not compiler options. They can only be used in the utilization tracking configuration file.
- If the **-qexemptconcurrentusers** entry is specified multiple times in the utilization tracking configuration file, all the specified instances are used. If other entries are specified multiple times, the value of the last one overrides previous ones.
- The compiler generates a message if you specify the above entries with different values for different users when using more than one utilization tracking configuration file. You must modify the entries to keep them consistent, or make sure all compiler users use a single utilization configuration file.

Product information

-qprodId=*product_identifier_string*
Indicates the unique product identifier string.

-qprodVer=*product_version*
Indicates the product version.

-qprodRel=*product_release*
Indicates the product release.

-qprodName=*product_name*
Indicates the product name.

-qconcurrentusagescope=prod | ver | rel
Specifies the level at which concurrent users are counted, and their numbers are limited. The suboptions are as follows:

- **prod** indicates the product level.
- **ver** indicates the version level.
- **rel** indicates the release level.

Default: **-qconcurrentusagescope=prod**

Tracking configuration

-qmaxconcurrentusers=*number*

Specifies the maximum number of concurrent users. It is the number of Concurrent User licenses that you have purchased for the product. When the utilization reporting tool generates a report from the usage file, it determines whether your compiler usage in your organization has exceeded this maximum number of concurrent users.

Note: You must update this entry to reflect the actual number of Concurrent User licenses that you have purchased.

Default: 0

-qexemptconcurrentusers ="*user_account_info_1* **[|** *user_account_info_2* **| ... |** *user_account_info_n***]"**

Specify exempt users who have their own Authorized User license. Exempt users can have as many concurrent invocations of the compiler as they want, without affecting the number of Concurrent User licenses available in your organization. When the utilization reporting tool generates a usage report, it does not include such users in the count of concurrent users.

user_account_info can be any combination of the following items:

- name(*user_name*)
- uid(*user_ID*)
- host(*host_name*)

Users whose information matches the specified criteria are considered exempt users. For example, to indicate that *user1@host1* and *user2@host1* are exempt users, you can specify this entry in either of the following forms:

-
  ```
  -qexemptconcurrentusers="name(user1)host(host1)"
  -qexemptconcurrentusers="name(user2)host(host1)"
  ```

-
  ```
  -qexemptconcurrentusers="name(user1)host(host1) | name(user2)host(host1)"
  ```

For *user_name*, *user_ID*, and *host_name*, you can also use a list of user names, user IDs, or hostnames separated by a space within the parentheses. For example:

```
-qexemptconcurrentusers="name(user1 user2)host(host1)"
```

This is equivalent to the previous examples.

Note: Specifying this entry does not exempt users from compiler utilization tracking. It only exempts them from being counted as concurrent users. To optimize utilization tracking performance, the format of the specified value is not validated until the report is produced. For more information about counting concurrent users, see "Concurrent user considerations" on page 55.

-qqualhostname | -qnoqualhostname

Specifies whether host names that are captured in usage files and then listed in compiler usage reports are qualified with domain names.

If all compiler usage within your organization is on machines within a single domain, you can reduce the size of the usage files by using **-qnoqualhostname** to suppress domain name qualification.

Default: **-qqualhostname**, which means the host names are qualified with domain names.

-qenabletracking | -qnoenabletracking

Enables or disables utilization tracking.

Default: **-qnoenabletracking**, which means utilization tracking is disabled.

`-qusagefileloc=``directory_or_ file_name`

Specifies the location of the usage file.

By default, a `.cuf` file is automatically created for each user in their home directory. You can set up a central location where the files for each user can be created. For more information, see "Usage file location" on page 56.

The following rules apply when you specify this entry:

- If a file name is specified, it must have the `.cuf` extension. If the file is a symlink, it must point to a file with the`.cuf` extension. If the specified file does not exist, a `.cuf` file is created, along with any parent directories that do not already exist.
- If a directory is specified, there must be exactly one file with the `.cuf` extension in the directory. A new file is not created in this case.
- The path of the specified directory can be a relative or an absolute path. Relative paths are relative to the compiler user's current working directory.

Note: If a compiler user cannot access the file, for example, because of insufficient permissions to use or create the file, the compiler generates a message and the compilation continues.

You can use the following variables for this option:

- *$HOME* for the user's home directory. This allows each user to have a `.cuf` file in their home directory or a subdirectory of their home directory.
- *$USER* or *$LOGNAME* for the user's login user name. You can use this variable to create a `.cuf` file for each user and include the user's login name in the name of the `.cuf` file or in the name of a parent directory.
- *$HOSTNAME* for the name of the host on which the compiler runs. This can be useful when you track utilization across different hosts.

`-qfileaccessmaxwait=``number_of_milliseconds`

Specifies the maximum number of milliseconds to wait for accessing the usage file.

Note: This entry is used to account for unusual circumstances where the system is under extreme heavy load and there is a delay in accessing the usage file.

Default: 3000 milliseconds

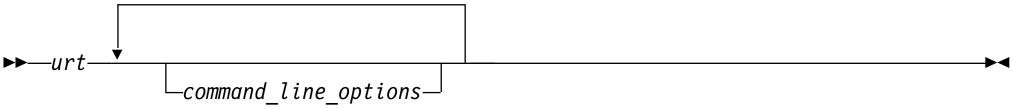
Notes:

- These entries are not compiler options. They can only be used in the utilization tracking configuration file.
- If the **-qexemptconcurrentusers** entry is specified multiple times in the utilization tracking configuration file, all the specified instances are used. If other entries are specified multiple times, the value of the last one overrides previous ones.

Understanding the utilization reporting tool

You can use the utilization reporting tool to generate compiler usage reports from the information in one or more usage files, and optionally prune the usage files when you generate the reports.

The tool is located in the `/opt/ibmurt/1.2/bin` directory. You can use the **urt** command to invoke it. The syntax of the **urt** command is as follows:



The generated report is displayed on the standard output. You can direct the output to a file if you want to keep the report.

Command-line options control how usage reports are generated. For more information about the options, see "Utilization reporting tool command-line options."

A default configuration file `ibmurt.cfg` is provided in the `/opt/ibmurt/1.2/config` directory. Entries in this file take the same form as the command-line options and have the same effect. You can also create additional configuration files and use the **-qconfigfile** option to specify their names.

You can specify the options in one or more of the following places:

- The default configuration file
- The additional configuration file specified with **-qconfigfile**
- The command line

The utilization reporting tool uses the options in the default configuration file before it uses the options on the command line. When it encounters a **-qconfigfile** option on the command line, it reads the options in the specified configuration file and puts them on the command line at the place where the **-qconfigfile** option is used.

If an option is specified multiple times, the last specification that the tool encounters takes effect. Exceptions are **-qconfigfile** and **-qsameuser**. For these two options, all specifications take effect.

Utilization reporting tool command-line options

The utilization reporting tool command-line options control the generation of the compiler utilization report.

Use these command-line options to modify the details of your compiler utilization report.

`-qreporttype=detail | maxconcurrent`

Specifies the type of the usage report to generate.

- **detail** specifies that all invocations of the compiler are listed in the usage report. With this suboption, you can get a detailed report, in which the invocations that have exceeded the allowed maximum number of concurrent users are indicated.

- **maxconcurrent** specifies that only the compiler invocations that have exceeded the allowed maximum number of concurrent users are listed. With this suboption, you can get a compact report, which does not list those invocations within the maximum number of allowed concurrent users.

Note: The allowed maximum number of concurrent users is specified with the **-qmaxconcurrentusers** entry in the utilization tracking configuration file.

Default: **-qreporttype=maxconcurrent**.

-qrptmaxrecords=*num* **|** **nomax**

Specifies the maximum number of records to list in the report for each product. *num* must be a positive integer.

Default: **-qrptmaxrecords=nomax**, which means all the records are listed.

-qusagefileloc=*directory_or_file_name*

Specifies the location of the usage files for report generation or pruning. It can be a list of directories or file names, or both, separated by colons.

The following rules apply when you specify this option:

- If one or more directories are specified, all files with the `.cuf` extension in those directories are used. Subdirectories can also be searched by using the **-qmaxsubdirs** option.
- The path of the specified directory can be relative or absolute. Relative paths are relative to the compiler user's current working directory.
- A symlink does not require the `.cuf` extension but the file to which it points must have that extension.

Note:

- The **-qusagefileloc** entry in the utilization tracking configuration file tells the compiler which usage files to use for recording compiler utilization. This **-qusagefileloc** option tells the utilization reporting tool where to find those usage files.

Default: *.:$HOME*, which means the utilization reporting tool looks for usage files in your current working directory and your home directory.

-qmaxsubdirs=*num* **|** **nomax**

Specifies whether to search subdirectories for usage files, and how many levels of subdirectories to search. *num* must be a non-negative integer.

If **nomax** is specified, all the subdirectories are searched. If 0 is specified, no subdirectories are searched.

Default: 0.

-qconfigfile=*file_path*

Specifies the user defined configuration file that you want to use.

For more information about how the utilization reporting tool uses the configuration file, see "Understanding the utilization reporting tool" on page 64.

Note: If you specify this option multiple times, all specified instances are used.

-qsameuser=*user_account_info*

Specifies different user accounts that belong to the same compiler user. Use this option when a user accesses the compiler from more than one user ID or machine to avoid having that user reported as multiple users. Invocations of the compiler by these different accounts are counted as a single user instead of multiple different users.

user_account_info can be any combination of the following items:

- name(*user_name*)
- uid(*user_ID*)
- host(*host_name*)

There are two ways to pass these rules to the utilization reporting tool. You can supply specific lists of the *user_name*s, *user_ID*s or*host_name*s that are shared by the same user or you can use a more generic (=) syntax.

For example, to indicate that *user1* and *user2* are both user names belonging to the same person who uses the compiler on the *host1* machine, use the syntax in which you specify these user names and the host name explicitly:

```
-qsameuser="name(user1)host(host1) | name(user2)host(host1)"
```

or

```
-qsameuser="name(user1 user2)host(host1)"
```

Both of these examples use specific user names and host names to indicate accounts that belong to the same user, but they do so in slightly different ways. The first example uses a vertical bar to separate the different user accounts that belong to this user, while the second example uses a list of user names within the parentheses instead of repeating the same host information twice. They both convey the same account information, but the second example is more concise.

As an example of the more generic (=) syntax, you can indicate that all user accounts with the same user name and uid belong to the same user as follows:

```
-qsameuser="name(=)uid(=)"
```

With this option, you are not specifying specific user names or uids as you did in the previous example. User accounts that have the same user name and uid are considered as belonging to the same user, regardless of what the specific user names and uids are, and regardless of what the host name is. This establishes a general rule that applies to all accounts in your organization instead of specific ones.

The following rules apply when you specify this option:

- Each instance of the **-qsameuser** option must use either the list or generic (=) syntax. You cannot combine them in the same instance of the option but you can use multiple instances of the **-qsameuser** option to refine the report.
- The utilization reporting tool matches the user information based on the order that the **-qsameuser** option values are specified. Once it finds a match it stops matching the same user information against any subsequent options.

  The following examples illustrate the differences:

  – If you specify the **-qsameuser** option as follows:

    ```
    -qsameuser="name(user1)" -qsameuser="uid(=)"
    ```

    Specifying the **-qsameuser** option in this order means that user accounts with the user name *user1* matches the first option and is not evaluated against the second option. User accounts *user1* and *user2* are not considered the same user even if they have the same *uid*.

- If you specify the **-qsameuser** option as follows:

  ```
  -qsameuser="uid(=)" -qsameuser="name(user1)"
  ```

  Specifying the **-qsameuser** option in this order means that user accounts with the same *uid* are always considered to be the same user, and in addition, any user accounts with a user name of *user1* should be considered belonging to the same user even if they do not match by *uid*.

Note: Specifying this option does not prevent user information from being listed in the usage report. For more information about concurrent users, see "Concurrent user considerations" on page 55.

-qadjusttime=*time_adjustments*

Adjusts the time that have been recorded in the usage files for the specified machines. *time_adjustments* is a list of entries with the format of *machine name* + | - *number of seconds*, separated by colons.

The following rules apply when you use this option:

- An entry of the form *machine name* + *number of seconds* causes the specified number of seconds to be added to the start and end times of any invocations recorded for the specified machine.
- An entry of the form *machine name* - *number of seconds* causes the specified number of seconds to be subtracted from the start and end times of any invocations recorded for the specified machine.

For example:

```
-qadjusttime="hostA+5:hostB-3"
```

Five seconds are added to the start and end times of the invocations on `hostA`, and three seconds are subtracted from the start and end times of the invocations on `hostB`.

Only use this option if the usage files contain utilization information from two or more machines, and time is not synchronized across those machines. The adjustments specified by this option compensate for the lack of synchronization

Notes:

- The specified adjustments are only used for the current run of the **urt** command. Specifying this option does not change the invocation information recorded in the usage files.
- Do not specify the same machine name more than once with this option.

-qusagefilemaxage=*number_of_days* **|** **nomax**

Prunes the usage files by removing all invocations older than the specified number of days.

Every usage file specified by the **-qusagefileloc** option is pruned. The usage report contains this information to indicate the number of records that have been pruned.

Default: **-qusagefilemaxage=nomax**, which means no pruning is performed.

-qusagefilemaxsize=*number_of_MB* **|** **nomax**

Prunes the usage files to keep them under the specified size. It prunes the files by removing the oldest invocations.

Every usage file specified by the **-qusagefileloc** option is pruned. The usage report contains this information to indicate the number of records that have been pruned.

Default: **-qusagefilemaxsize=nomax**, which means no pruning is performed.

-qtimesort=ascend | descend

Specifies the chronological order in which the usage report information is sorted.

- Specifying **ascend** means new information is listed after the older information.
- Specifying **descend** means the newest information is at the top of the report.

Default: **-qtimesort=ascend**.

Generating usage reports

You can use the utilization reporting tool to generate compiler usage reports based on the usage information stored in the usage files.

To generate a report, use the command-line options or the **urt** configuration file to specify how you want a report to be generated. For more information about these options, see "Utilization reporting tool command-line options" on page 64.

Notes:

- You can set up a cron service to run the utilization reporting tool on a regular basis. If the usage files from which the tool generate reports need to be copied to the machine where the tool is running, you can also automate this copying task with cron.
- To reduce contention for read and write access to the usage files, do not run the tool or copy the usage files when the compiler is being used.

The generated report is displayed on the standard output. You can direct the output to a file if you want to keep the report.

After a usage report is generated, the utilization reporting tool uses the following exit codes to indicate the compliance status of your compiler license:

- Exit code =`"1"`.

  The utilization reporting tool has detected that the number of Concurrent User license entitlements specified in the **-qmaxconcurrentusers** entry in the utilization tracking configuration file has been exceeded. See the generated report for details and contact your IBM representative to purchase additional Concurrent User licenses.
- Exit code =`"0"`.

  The compiler utilization is within the number of Concurrent User license entitlements specified.

For more information about the **urt** command, see "Understanding the utilization reporting tool" on page 64.

Understanding usage reports

You can use the report that the utilization report tool generates to analyze the compiler usage in your organization.

The report has a REPORT SUMMARY section that lists the following information:

1. The date and time when the report was generated.
2. The `.cuf` file or a list of all `.cuf` files used to generate the report.
3. The options that were passed to the **urt** command, with default values for any unspecified options.
4. Possible messages categorized as ERROR, WARNING, or INFO. For detailed information about possible messages, see "Diagnostic messages from utilization tracking and reporting" on page 72.

After the summary section, there is a REPORT DETAILS section for each compiler version. This section lists all of the compiler invocations recorded in the usage files. The content of these sections varies depending on the report type that you have specified. For detailed information about the report types, see **-qreporttype**.

Here are the sample reports generated with the two different report types:

Sample 1: A sample report generated with **-qreporttype=detail**

```
REPORT SUMMARY
--------------

DATE: 12/18/15      TIME: 01:30:24

OPTIONS USED (* indicates that a default value was used):

reporttype=detail
maxsubdirs=0
configfile="/opt/ibmurt/1.2/config/ibmurt.cfg"
rptmaxrecords=nomax
*adjusttime=
usagefileloc="/home/testrun/ibmxlcompiler.cuf"
*sameuser=
timesort=ascend
usagefilemaxsize=nomax
usagefilemaxage=nomax


FILES USED:

/home/testrun/ibmxlcompiler.cuf

REPORT DETAILS
--------------

USAGE INFORMATION FOR PRODUCT: IBM XL C for AIX 13.1.3

Max. Concurrent Users Exceeded? : *** YES ***

Max. Concurrent Users Allowed: 1          Max. Concurrent Users Recorded: 5
Exempt Users:

Product invocations:

Start Time         End Time           User               Number of Concurrent Users
------------------ ------------------ -----------------   --------------------------
12/17/15 16:56:44  12/17/15 16:57:13  user1@host1.ibm.com 1
12/18/15 00:58:29  12/18/15 00:58:32  user2@host2.ibm.com 1
12/18/15 01:16:01  12/18/15 01:16:02  user3@host3.ibm.com 5 ( exceeds max. allowed)
12/18/15 01:16:02  12/18/15 01:16:26  user2@host2.ibm.com 5 ( exceeds max. allowed)
12/18/15 01:16:08  12/18/15 01:16:08  user3@host2.ibm.com 5 ( exceeds max. allowed)
12/18/15 01:16:12  12/18/15 01:16:12  user2@host1.ibm.com 5 ( exceeds max. allowed)
12/18/15 01:16:24  12/18/15 01:16:28  user1@host2.ibm.com 5 ( exceeds max. allowed)
12/18/15 01:26:11  12/18/15 01:27:46  user3@host3.ibm.com 2 ( exceeds max. allowed)
12/18/15 01:26:27  12/18/15 01:27:46  user1@host1.ibm.com 2 ( exceeds max. allowed)
```

```
12/18/15 01:29:59  12/18/15 01:30:00  user2@host1.ibm.com 1
12/18/15 01:30:00  12/18/15 01:30:00  user2@host2.ibm.com 3 ( exceeds max. allowed)
12/18/15 01:30:14  12/18/15 01:30:15  user3@host1.ibm.com 3 ( exceeds max. allowed)
12/18/15 01:30:14  12/18/15 01:30:14  user2@host2.ibm.com 3 ( exceeds max. allowed)
```

Sample 2: A sample report generated with **-qreporttype=maxconcurrent**

```
REPORT SUMMARY
--------------

DATE: 12/18/15      TIME: 01:32:53

OPTIONS USED (* indicates that a default value was used):

reporttype=maxconcurrent
maxsubdirs=0
configfile="/opt/ibmurt/1.2/config/ibmurt.cfg"
rptmaxrecords=nomax
*adjusttime=
usagefileloc="/home/testrun/ibmxlcompiler.cuf"
*sameuser=
timesort=ascend
usagefilemaxsize=nomax
usagefilemaxage=nomax


FILES USED:

/home/testrun/ibmxlcompiler.cuf

REPORT DETAILS
--------------

USAGE INFORMATION FOR PRODUCT: IBM XL C for AIX 13.1.3

Max. Concurrent Users Exceeded? : *** YES ***

Max. Concurrent Users Allowed: 1          Max. Concurrent Users Recorded: 5

Exempt Users:

Dates and times where usage exceeded the maximum allowed:

Date           Time        Number of Concurrent Users    Users
------------   ----        --------------------------    -----
12/18/15       01:16:01    5                             user3@host3.ibm.com
                                                         user2@host2.ibm.com
                                                         user3@host2.ibm.com
                                                         user2@host1.ibm.com
                                                         user1@host2.ibm.com
12/18/15       01:16:02    5                             user3@host3.ibm.com
                                                         user2@host2.ibm.com
                                                         user3@host2.ibm.com
                                                         user2@host1.ibm.com
                                                         user1@host2.ibm.com
12/18/15       01:16:08    5                             user3@host3.ibm.com
                                                         user2@host2.ibm.com
                                                         user3@host2.ibm.com
                                                         user2@host1.ibm.com
                                                         user1@host2.ibm.com
12/18/15       01:16:12    5                             user3@host3.ibm.com
                                                         user2@host2.ibm.com
                                                         user3@host2.ibm.com
                                                         user2@host1.ibm.com
                                                         user1@host2.ibm.com
12/18/15       01:16:24    5                             user3@host3.ibm.com
                                                         user2@host2.ibm.com
```

```
                                                    user3@host2.ibm.com
                                                    user2@host1.ibm.com
                                                    user1@host2.ibm.com
12/18/15       01:26:11    2                        user3@host3.ibm.com
                                                    user1@host1.ibm.com
12/18/15       01:26:27    2                        user3@host3.ibm.com
                                                    user1@host1.ibm.com
12/18/15       01:30:00    3                        user2@host2.ibm.com
                                                    user2@host1.ibm.com
                                                    user3@host1.ibm.com
12/18/15       01:30:14    3                        user2@host2.ibm.com
                                                    user2@host1.ibm.com
                                                    user3@host1.ibm.com
12/18/15       01:30:14    3                        user2@host2.ibm.com
                                                    user2@host1.ibm.com
                                                    user3@host1.ibm.com
```

Note: There are circumstances under which an end time might not be recorded. These might include:

- There was a major failure of the compiler, for example, power loss during a compilation.
- The invocation had not ended at the time when the report was generated, or at the time when the usage file was being copied.
- The permission to write to the usage file was revoked at some time before the end time of the invocation was recorded.

An invocation with no end time recorded is not included in the count of concurrent users.

Pruning usage files

Usage files grow with each compiler invocation. You can prune the usage files with the utilization report tool.

When you generate a usage report, you can specify the following two options to optionally prune the usage files:

- **-qusagefilemaxage**: Removes the invocations older than the specified number of days. For example, to remove all entries in the usage files older than 30 days, use the following command:

  ```
  urt -qusagefilemaxage=30
  ```
- **-qusagefilemaxsize**: Removes the oldest invocations to keep the usage files under the specified size. For example, to remove the oldest invocations to keep the usage files under 30 MB, use the following command:

  ```
  urt -qusagefilemaxsize=30
  ```

When usage files are pruned, the usage report includes an information message that indicates the number of records that have been pruned. If you want to keep the generated report after the files are pruned, you can redirect the output to a file.

To control the size of the usage files, you can prune the usage files on a regular basis. You can create a cron job to do this automatically.

If you do not have the utilization reporting tool installed on each machine where the usage files are located, you have the following options:

- Export the file system from each machine where the usage files exist and mount it on the machine where the utilization reporting tool is installed. Then run the tool to prune the usage files on the mounted network file system.

- After copying the usage files to the machine where the utilization reporting tool is installed, delete the files and use new usage files to capture any subsequent compiler invocations. This approach might also speed up the report generation because the utilization reporting tool is not accessing the usage files remotely and it is not spending time pruning the usage files.

Pruning usage files might slow down the usage report generation process, especially when the number or the size of the usage files is large. If you do not want to prune the files every time you generate reports, you can set the values for the **-qusagefilemaxage** and **-qusagefilemaxsize** options as follows:

- If you generate the report daily, you can specify these two options with very high values so pruning does not occur. The default value **nomax** can be used in this case.
- You can set appropriate values for these two options and use a separate cron job to prune the usage files weekly.

Note: To reduce contention for read and write access to the usage files, do not run the utilization report tool or copy the usage files when the compiler is being used.

Diagnostic messages from utilization tracking and reporting

The compiler generates diagnostic messages to indicate utilization tracking issues. These messages can help you to fix the associated problems.

For example:

```
Utilization tracking configuration file could not be read due to
insufficient permissions.
```

This message indicates that you need read access for utilization tracking configuration file.

When the utilization reporting tool is used to generate usage reports or prune usage files, it also generates diagnostic messages. For example:

```
Unrecognized option -qmaxsubdir.
```

This message indicates that you have specified a wrong option.

Note: Possible error, warning, or information messages are also included in the compiler usage report generated by the tool.

Tracking compiler usage with Software License Metric Tags logging

In addition to the utilization reporting tool, 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.

About this task

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 for AIX, V13.1.3, SLM Tags logging is provided for evaluation purposes only, and logging is enabled only when the **-qxflag=slmtags** compiler option is specified to invoke the license metric logging. When logging is enabled, the compiler logs compiler license usage in the SLM Tags format, to a file 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.

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

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

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

   ```
   24461,ae80b54aac3143fdb0248ec565554539
   9204,ae80b54aac3143fdb0248ec565554539
   7531,ae80b54aac3143fdb0248ec565554539
   ```
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 by functional category, followed by detailed descriptions of the individual options.

Related information

- “Specifying compiler options” on page 5
- “Reusing GNU C compiler options with gxlc” on page 11

Summary of compiler options by functional category

The XL C options available on the AIX 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 76
- “Language element control” on page 77
- “Floating-point and integer control” on page 78
- “Error checking and debugging” on page 81
- “Listings, messages, and compiler information” on page 84
- “Optimization and tuning” on page 85
- “Object code control” on page 79
- “Linking” on page 89
- “Portability and migration” on page 90
- “Compiler customization” on page 91
- “Deprecated options” on page 91

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 10. Compiler output options

Option name	Equivalent pragma name	Description
“-c” on page 114	None.	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 115	None.	When used in conjunction with the **-E** or **-P** options, preserves or removes comments in preprocessed output.

Table 10. Compiler output options (continued)

Option name	Equivalent pragma name	Description
“-E” on page 136	None.	Preprocesses the source files named in the compiler invocation, without compiling, and writes the output to the standard output.
“-G” on page 163	None.	Generates a shared object enabled for runtime linking.
“-qmakedep, -M” on page 226	None.	Produces the dependency files that are used by the **make** tool for each source file.
“-MF” on page 231	None.	Specifies the name or location for the dependency output files that are generated by the **-qmakedep** or **-M** option.
“-qmkshrobj” on page 233	None.	Creates a shared object from generated object files.
“-o” on page 235	None.	Specifies a name for the output object, assembler, executable, or preprocessed file.
“-P” on page 244	None.	Preprocesses the source files named in the compiler invocation, without compiling, and creates an output preprocessed file for each input file.
“-S” on page 271	None.	Generates an assembler language file for each source file.
“-qshowmacros” on page 276	None.	Emits macro definitions to preprocessed output.
“-qtimestamps” on page 307	None.	Controls whether or not implicit time stamps are inserted into an object file.

Input control

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

Table 11. Compiler input options

Option name	Equivalent pragma name	Description
“-I” on page 172	None.	Adds a directory to the search path for include files.
“-qidirfirst” on page 173	#pragma options idirfirst	Searches for user included files in directories that are specified by the **-I** option before searching any other directories.

Table 11. Compiler input options (continued)

Option name	Equivalent pragma name	Description
"-qinclude" on page 176	None.	Specifies additional header files to be included in a compilation unit, as though the files were named in an `#include` statement in the source file.
"-qsourcetype" on page 287	None.	Instructs the compiler to treat all recognized source files as a specified source type, regardless of the actual file name suffix.
"-qstdinc" on page 292	#pragma options stdinc	Specifies whether the standard include directories are included in the search paths for system and user header files.

Language element control

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

Table 12. Language element control options

Option name	Equivalent pragma name	Description
"-qaltivec" on page 101	None	Enables the compiler support for vector data types and operators.
"-qasm" on page 105	None	Controls the interpretation and subsequent generation of code for assembler language extensions.
"-qcpluscmt" on page 123	None.	Enables recognition of C++-style comments in C source files.
"-D" on page 126	None.	Defines a macro as in a `#define` preprocessor directive.
"-qdfp" on page 131	None.	Enables compiler support for decimal floating-point types and literals.
"-qdigraph" on page 132	#pragma options digraph	Enables recognition of digraph key combinations to represent characters that are not found on some keyboards. Digraph key combinations include <:, <%, and so on.
"-qdollar" on page 133	#pragma options dollar	Allows the dollar-sign ($) symbol to be used in the names of identifiers.
"-qignprag" on page 175	#pragma options ignprag	Instructs the compiler to ignore certain pragma statements.

Table 12. Language element control options (continued)

Option name	Equivalent pragma name	Description
"-qkeyword" on page 203	None.	Controls whether the specified name is treated as a keyword or as an identifier whenever it appears in your program source.
"-qlanglvl" on page 206	#pragma options langlvl, #pragma langlvl	Determines whether source code and compiler options should be checked for conformance to a specific language standard, or subset or superset of a standard.
"-qlonglong" on page 223	#pragma options long long	Allows IBM `long long` integer types in your program.
"-qmacpstr" on page 224	#pragma options macpstr	Converts Pascal string literals (prefixed by the `\p` escape sequence) into null-terminated strings in which the first byte contains the length of the string.
"-qmbcs, -qdbcs" on page 230	#pragma options mbcs, #pragma options dbcs	Enables support for multibyte character sets (MBCS) and Unicode characters in your source code.
"-qtabsize" on page 304	None.	Sets the default tab length, for the purposes of reporting the column number in error messages.
"-qtrigraph" on page 310	None.	Enables the recognition of trigraph key combinations to represent characters not found on some keyboards.
"-U" on page 313	None.	Undefines a macro defined by the compiler or by the **-D** compiler option.
"-qutf" on page 319	None.	Enables recognition of UTF literal syntax.

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	Equivalent pragma name	Description
"-qbitfields" on page 111	None.	Specifies whether bit fields are signed or unsigned.

Table 13. Floating-point and integer control options (continued)

Option name	Equivalent pragma name	Description
"-qchars" on page 118	#pragma options chars, #pragma chars	Determines whether all variables of type `char` is treated as `signed` or `unsigned`.
"-qenum" on page 137	#pragma options enum, #pragma enum	Specifies the amount of storage occupied by enumerations.
"-qfloat" on page 146	#pragma options float	Selects different strategies for speeding up or improving the accuracy of floating-point calculations.
"-qldbl128, -qlongdouble" on page 212	#pragma options ldbl128	Increases the size of `long double` types from 64 bits to 128 bits.
"-qlonglit" on page 222	None.	In 64-bit mode, when determining the implicit types for integer literals, the compiler behaves as if an `l` or `L` suffix were added to integral literals with no suffix or with a suffix consisting only of `u` or `U`.
"-qstrict" on page 294	#pragma options [no]strict #pragma option_override (*function_name*, "opt (*suboption_list*)")	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 332	None.	Specifies the rounding mode for the compiler to use when evaluating constant floating-point expressions at compile time.

Object code control

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

Table 14. Object code control options

Option name	Equivalent pragma name	Description
"-q32, -q64" on page 94	None.	Selects either 32-bit or 64-bit compiler mode.
"-qalloca, -ma" on page 100	#pragma alloca	Provides an inline definition of system function `alloca` when it is called from source code that does not include the `alloca.h` header.
"-qconcurrentupdate" on page 123	None.	Supports updating the operating system while the kernel is running.

Table 14. Object code control options (continued)

Option name	Equivalent pragma name	Description
"-qexpfile" on page 141	None.	When used together with the **-qmkshrobj** or **-G** option, saves all exported symbols in a designated file.
"-qinlglue" on page 188	#pragma options inlglue	When used with **-O2** or higher optimization, inlines glue code that optimizes external function calls in your application.
"-qpic" on page 254	None.	Generates position-independent code suitable for use in shared libraries.
"-qppline" on page 255	None.	When used in conjunction with the **-E** or **-P** options, enables or disables the generation of `#line` directives.
"-qproto" on page 262	#pragma options proto	Specifies the linkage conventions for passing floating-point arguments to functions that have not been prototyped.
"-qreserved_reg" on page 265	None.	Indicates that the given list of registers cannot be used during the compilation except as a stack pointer, frame pointer or in some other fixed role.
"-qro" on page 267	#pragma options ro, #pragma strings	Specifies the storage type for string literals.
"-qroconst" on page 268	#pragma options roconst	Specifies the storage location for constant values.
"-qroptr" on page 270	None.	Specifies the storage location for constant pointers.
"-s" on page 271	None.	Strips the symbol table, line number information, and relocation information from the output file.
"-qsaveopt" on page 272	None.	Saves the command-line options used for compiling a source file, the user's configuration file name and the options specified in the configuration files, the version and level of each compiler component invoked during compilation, and other information to the corresponding object file.

Table 14. Object code control options (continued)

Option name	Equivalent pragma name	Description
"-qstackprotect" on page 291	None.	Provides protection against malicious input data or programming errors that overwrite or corrupt the stack.
"-qstatsym" on page 292	None.	Adds user-defined, nonexternal names that have a persistent storage class, such as initialized and uninitialized static variables, to the symbol table of the object file.
"-qtbtable" on page 305	#pragma options tbtable	Controls the amount of debugging traceback information that is included in the object files.
"-qthreaded" on page 306	None.	Indicates to the compiler whether it must generate threadsafe code.
"-qtls" on page 307	None.	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.
"-qweakexp" on page 328	None.	When used with the **-qmkshrobj** or **-G** option, includes or excludes global symbols marked as weak from the export list generated when you create a shared object.
"-qweaksymbol" on page 329	None.	Enables the generation of weak symbols.
"-qxcall" on page 330	None.	Generates code to treat static functions within a compilation unit as if they were external functions.

Error checking and debugging

The options in this category allow you to detect and correct problems in your source code. In some cases, these options can alter your object code, increase your compile time, or introduce runtime checking that can slow down the execution of your application. The option descriptions indicate how extra checking can impact performance.

To control the amount and type of information you receive regarding the behavior and performance of your application, consult the options in "Listings, messages, and compiler information" on page 84.

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

Table 15. Error checking and debugging options

Option name	Equivalent pragma name	Description
"-# (pound sign)" on page 93	None.	Previews the compilation steps specified on the command line, without actually invoking any compiler components.
"-qcheck" on page 119	#pragma options check	Generates code that performs certain types of runtime checking.
"-qdbgfmt" on page 129	None	Specifies the format for the debugging information in object files.
"-qdbxextra" on page 130	#pragma options dbxextra	When used with the **-g** option, specifies that debugging information is generated for unreferenced `typedef` declarations, `struct`, `union`, and `enum` type definitions.
"-qdpcl" on page 134	None.	Generates symbols that tools based on the IBM Dynamic Probe Class Library (DPCL) can use to see the structure of an executable file.
"-qextchk" on page 141	#pragma options extchk	Generates link-time type checking information and checks for compile-time consistency.
"-qflttrap" on page 151	#pragma options flttrap	Determines what types of floating-point exceptions to detect at run time.
"-qformat" on page 155	None.	Warns of possible problems with string input and output format specifications.
"-qfullpath" on page 156	#pragma options fullpath	When used with the **-g** or **-qlinedebug** option, this option records the full, or absolute, path names of source and include files in object files compiled with debugging information, so that debugging tools can correctly locate the source files.
"-qfunctrace" on page 158	None.	Calls the tracing routines to trace the entry and exit points of the specified functions in a compilation unit.
"-g" on page 160	None.	Generates debugging information for use by a symbolic debugger, and makes the program state available to the debugging session at selected source locations.

Table 15. Error checking and debugging options (continued)

Option name	Equivalent pragma name	Description
"-qhalt" on page 165	#pragma options halt	Stops compilation before producing any object, executable, or assembler source files if the maximum severity of compile-time messages equals or exceeds the severity you specify.
"-qhaltonmsg" on page 166	None.	Stops compilation before producing any object files, executable files, or assembler source files if a specified error message is generated.
"-qheapdebug" on page 167	None.	Enables debug versions of memory management functions.
"-qinfo" on page 178	#pragma options info, #pragma info	Produces or suppresses groups of informational messages.
"-qinitauto" on page 186	#pragma options initauto	Initializes uninitialized automatic variables to a specific value, for debugging purposes.
"-qkeepparm" on page 202	None.	When used with **-O2** or higher optimization, specifies whether procedure parameters are stored on the stack.
"-qlinedebug" on page 216	None.	Generates only line number and source file name information for a debugger.
"-qmaxerr" on page 228	None.	Stops compilation when the number of error messages of a specified severity level or higher reaches a specified number.
"-qoptdebug" on page 239	None.	When used with high levels of optimization, produces files containing optimized pseudocode that can be read by a debugger.
"-qsymtab" on page 301	None.	Determines the information that appears in the symbol table.
"-qsyntaxonly" on page 302	None.	Performs syntax checking without generating an object file.
"-qwarn64" on page 327	None.	Enables checking for possible data conversion problems between 32-bit and 64-bit compiler modes.

Listings, messages, and compiler information

The options in this category allow 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 81 to provide a more robust overview of your application when checking for errors and unexpected behavior.

Table 16. Listings and messages options

Option name	Equivalent pragma name	Description
"-qattr" on page 108	#pragma options attr	Produces a compiler listing that includes the attribute component of the attribute and cross-reference section of the listing.
"-qflag" on page 145	#pragma options flag	Limits the diagnostic messages to those of a specified severity level or higher.
"-qhelp" on page 169	None.	Displays the man page of the compiler.
"-qlist" on page 217	#pragma options list	Produces a compiler listing file that includes object and constant area sections.
"-qlistfmt" on page 218	None.	Creates a report in XML or HTML format to help you find optimization opportunities.
"-qlistopt" on page 221	None.	Produces a compiler listing file that includes all options in effect at the time of compiler invocation.
"-qphsinfo" on page 253	None.	Reports the time taken in each compilation phase to standard output.
"-qprint" on page 259	None.	Enables or suppresses listings.
"-qreport" on page 263	None.	Produces listing files that show how sections of code have been optimized.
"-qshowinc" on page 275	#pragma options showinc	When used with **-qsource** option to generate a listing file, selectively shows user or system header files in the source section of the listing file.

Table 16. Listings and messages options (continued)

Option name	Equivalent pragma name	Description
“-qskipsrc” on page 280	None.	When a listing file is generated using the **-qsource** option, **-qskipsrc** can be used to determine whether the source statements skipped by the compiler are shown in the source section of the listing file. Alternatively, the **-qskipsrc=hide** option is used to hide the source statements skipped by the compiler.
“-qsource” on page 286	#pragma options source	Produces a compiler listing file that includes the source section of the listing and provides additional source information when printing error messages.
“-qsrcmsg” on page 290	#pragma options srcmsg	Adds the corresponding source code lines to diagnostic messages generated by the compiler.
“-qsuppress” on page 299	None.	Prevents specific informational or warning messages from being displayed or added to the listing file, if one is generated.
“-v, -V” on page 319	None.	Reports the progress of compilation, by naming the programs being invoked and the options being specified to each program.
“-qversion” on page 321	None.	Displays the version and release of the compiler being invoked.
“-w” on page 325	None.	Suppresses warning messages.
“-qxref” on page 330	#pragma options xref	Produces a compiler listing that includes the cross-reference component of the attribute and cross-reference section of the listing.

Optimization and tuning

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

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

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

In addition to the option descriptions in this section, consult the *XL C 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	Equivalent pragma name	Description
"-qaggrcopy" on page 95	None.	Enables destructive copy operations for structures and unions.
"-qalias" on page 96	None.	Indicates whether a program contains certain categories of aliasing or does not conform to C standard aliasing rules. The compiler limits the scope of some optimizations when there is a possibility that different names are aliases for the same storage location..
"-qarch" on page 102	None.	Specifies the processor architecture for which the code (instructions) should be generated.
"-qcache" on page 116	None.	Specifies the cache configuration for a specific execution machine.
"-qcompact" on page 122	#pragma options compact	Avoids optimizations that increase code size.
"-qdataimported, -qdatalocal, -qtocdata" on page 127	None.	Marks data as local or imported.
"-qdirectstorage" on page 133	None.	Informs the compiler that a given compilation unit may reference write-through-enabled or cache-inhibited storage.
"-qfdpr" on page 144	None.	Provides object files with information that the IBM Feedback Directed Program Restructuring (FDPR®) performance-tuning utility needs to optimize the resulting executable file.
"-qhot" on page 169	#pragma nosimd, #pragma novector	Performs high-order loop analysis and transformations (HOT) during optimization.
"-qignerrno" on page 174	#pragma options ignerrno	Allows the compiler to perform optimizations as if system calls would not modify `errno`.
"-qipa" on page 193	None.	Enables or customizes a class of optimizations known as interprocedural analysis (IPA).

Table 17. Optimization and tuning options (continued)

Option name	Equivalent pragma name	Description
"-qisolated_call" on page 199	#pragma options isolated_call, #pragma isolated_call	Specifies functions in the source file that have no side effects other than those implied by their parameters.
"-qlargepage" on page 211	None.	Takes advantage of large pages provided on POWER4 and higher systems, for applications designed to execute in a large page memory environment.
"-qlibansi" on page 214	#pragma options libansi	Assumes that all functions with the name of an ANSI C library function are in fact the system functions.
"-qlibmpi" on page 215	None.	Asserts that all functions with Message Passing Interface (MPI) names are in fact MPI functions and not a user function with different semantics.
"-qmaxmem" on page 229	#pragma options maxmem	Limits the amount of memory that the compiler allocates while performing specific, memory-intensive optimizations to the specified number of kilobytes.
"-qminimaltoc" on page 232	None.	Controls the generation of the table of contents (TOC), which the compiler creates for an executable file.
"-O, -qoptimize" on page 236	#pragma options optimize	Specifies whether to optimize code during compilation and, if so, at which level.
"-p, -pg, -qprofile" on page 243	None.	Prepares the object files produced by the compiler for profiling.
"-qpdf1, -qpdf2" on page 247	None.	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 256	None.	Inserts prefetch instructions automatically where there are opportunities to improve code performance.
"-qprocimported, -qproclocal, -qprocunknown" on page 260	#pragma options procimported, #pragma options proclocal, #pragma options procunkown	Marks functions as local, imported, or unknown.

Table 17. Optimization and tuning options (continued)

Option name	Equivalent pragma name	Description
"-qinline" on page 189	None.	Attempts to inline functions instead of generating calls to those functions, for improved performance.
"-qrestrict" on page 266	None.	Specifying this option is equivalent to adding the `restrict` keyword to the pointer parameters within the specified functions, except that you do not need to modify the source file.
"-qshowpdf" on page 277	None.	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 278	#pragma nosimd	Controls whether the compiler can automatically take advantage of vector instructions for processors that support them.
"-qsmallstack" on page 281	None.	Minimizes stack usage where possible. Disables optimizations that increase the size of the stack frame.
"-qsmp" on page 281	None.	Enables parallelization of program code.
"-qspeculateabsolutes" on page 288	None.	Works with the **-qtocmerge -bl:file** for non-IPA links and with the **-bl:file** for IPA links to disable speculation at absolute addresses.
"-qstrict" on page 294	#pragma options strict	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 299	None.	Prevents the compiler from performing induction (loop counter) variable optimizations. These optimizations may be unsafe (may alter the semantics of your program) when there are integer overflow operations involving the induction variables.
"-qtocmerge" on page 309	None.	Enables TOC merging to reduce TOC pointer loads and improves the scheduling of external loads.

Table 17. Optimization and tuning options (continued)

Option name	Equivalent pragma name	Description
-qtune	#pragma options tune	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.
"-qunroll" on page 314	#pragma options unroll, #pragma unroll	Controls loop unrolling, for improved performance.
"-qunwind" on page 317	None.	Specifies whether the call stack can be unwound by code looking through the saved registers on the stack.
"-qvisibility" on page 323	#pragma GCC visibility push, #pragma GCC visibility pop	Specifies the visibility attribute for external linkage entities in object files. The external linkage entities have the visibility attribute that is specified by the **-qvisibility** option if they do not get visibility attributes from pragma directives, explicitly specified attributes, or propagation rules.

Linking

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

Table 18. Linking options

Option name	Equivalent pragma name	Description
"-b" on page 109	None.	Sets special linker processing options. This option can be repeated.
"-bmaxdata" on page 112	None.	Sets the maximum size of the area shared by the static data (both initialized and uninitialized) and the heap.
"-brtl" on page 113	None.	Enables runtime linking for the output file. When you use **-brtl** with the **-l** option, the linker searches for a library with the suffix of *.so*, as well as of *.a*. Preference is given to *.so* over *.a* when libraries with the same name are present in the same directory.
"-qcrt" on page 124	None.	Specifies whether system startup files are to be linked.

Table 18. Linking options (continued)

Option name	Equivalent pragma name	Description
"-e" on page 135	None.	When used together with the **-qmkshrobj** option or **-G** option, specifies an entry point for a shared object.
"-f" on page 142	None.	Names a file that stores a list of object files for the compiler to pass to the linker.
"-L" on page 205	None.	Searches the directory path for library files specified by the **-l** option.
"-l" on page 204	None.	Searches for the specified library file. For static and dynamic linking, the linker searches for *libkey.a*. For runtime linking with the **-brtl** option, the linker searches for *libkey.so*, and then *libkey.a* if *libkey.so* is not found.
"-qlib" on page 213	None.	Specifies whether standard system libraries and XL C libraries are to be linked.
"-Z" on page 333	None.	Specifies a prefix for the library search path to be used by the linker.

Portability and migration

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

Table 19. Portability and migration options

Option name	Equivalent pragma name	Description
"-qalign" on page 98	#pragma options align, #pragma align	Specifies the alignment of data objects in storage, which avoids performance problems with misaligned data.
"-qgenproto" on page 164	None.	Produces prototype declarations from K&R function definitions or function definitions with empty parentheses, and displays them to standard output.
"-qupconv" on page 318	#pragma options upconv	Specifies whether the `unsigned` specification is preserved when integral promotions are performed.
"-qvecnvol" on page 320	None.	Specifies whether to use volatile or nonvolatile vector registers.

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	Equivalent pragma name	Description
"-qasm_as" on page 107	None.	Specifies the path and flags used to invoke the assembler in order to handle assembler code in an `asm` assembly statement.
"-B" on page 110	None.	Specifies substitute path names for XL C components such as the assembler, C preprocessor, and linker.
"-qc_stdinc" on page 125	None.	Changes the standard search location for the XL C and system header files.
"-F" on page 143	None.	Names an alternative configuration file or stanza for the compiler.
"-qpath" on page 245	None.	Specifies substitute path names for XL C components such as the compiler, assembler, linker, and preprocessor.
"-qoptfile" on page 241	None.	Specifies a file containing a list of additional command line options to be used for the compilation.
"-qspill" on page 289	#pragma options spill	Specifies the size (in bytes) of the register spill space, the internal program storage areas used by the optimizer for register spills to storage.
"-t" on page 303	None.	Applies the prefix specified by the **-B** option to the designated components.
"-W" on page 326	None.	Passes the listed options to a component that is executed during compilation.

Deprecated options

The compiler still accepts options that are listed in the following table. Options without an asterisk have been replaced by other options or environment variables that provide the same functionality. Options with an asterisk are obsolete, or can produce unexpected results and are not guaranteed to perform as previously documented. Use with discretion.

Table 21. Deprecated options

Option name	Replacement option
-Q	-qinline
-qansialias	-qalias=ansi

Table 21. Deprecated options (continued)

Option name	Replacement option
-qarch = ppc | ppc64 | ppcgr | ppc64gr | ppc64grsq	-qarch=pwr4
-qassert	-qalias
-qenablevmx	-qsimd
-qfloat=emulate*	
-qfold	-qfloat=fold
-qhsflt	-qfloat=hsflt
-qhssngl	-qfloat=hssngl
-qhot=simd | nosimd	-qsimd
-qinfo=private	-qreport
-qinfo=reduction	-qreport
-qipa=clonearch | noclonearch	-qtune
-qipa=cloneproc | nocloneproc	-qtune
-qipa=inline | noinline	-qipa -qinline | -qipa -qnoinline
-qipa=pdfname	-qpdf1=pdfname, -qpdf2=pdfname
-qlanglvl=[no]gnu_externtemplate	-qlanglvl=[no]externtemplate
-qmaf	-qfloat=maf
-qrrm	-qfloat=rrm
-qsmp= schedule=affinity	-qsmp=schedule=guided
-qsmp= nested_par | nonested_par	The “OMP_NESTED” on page 33 environment variable or “omp_set_nested” on page 626 function
-qspnans	-qfloat=spnans

Individual option descriptions

This section contains descriptions of the individual compiler options available in XL 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 options** *option_name* and/or **#pragma** ***name*** form of the directive, this is indicated.

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

Syntax
This section provides the syntax for the option, and where an equivalent

#pragma ***name*** is supported, the specific syntax for the pragma. Syntax for **#pragma options** *option_name* forms of the pragma is not provided, as this is normally identical to that of the option.

Note that you can also use the C99-style `_Pragma` operator form of any pragma; although this syntax is not provided in the option descriptions. For complete details on pragma syntax, see "Pragma directive syntax" on page 335

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 403

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 319

-q32, -q64

Category

Object code control

Pragma equivalent

None.

Purpose

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

Use the **-q32** and **-q64** options, along with the **-qarch** and **-qtune** compiler options, to optimize the output of the compiler to the architecture on which that output will be used.

Syntax

```
          ┌─32─┐
►►── -q───┴─64─┴─────────────────────────────────────────────────────►◄
```

Defaults

-q32

Usage

The **-q32** and **-q64** options override the compiler mode set by the value of the OBJECT_MODE environment variable, if it exists.

Predefined macros

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

Examples

To specify that the executable program `testing` compiled from `myprogram.c` is to run on a computer with a 32-bit Power architecture, enter:

```
xlc -o testing myprogram.c -q31 -qarch=ppc
```

Related information

- Specifying compiler options for architecture-specific compilation
- “-qarch” on page 102
- “-qtune” on page 310

-qaggrcopy

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Enables destructive copy operations for structures and unions.

Syntax

```
                         ┌─nooverlap─┐
►►── -q──aggrcopy──=─────┴─overlap───┴───────────────────────────────────────►◄
```

Defaults

-qaggrcopy=nooverlap

Parameters

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

Predefined macros

None.

-qalias

Category

Optimization and tuning

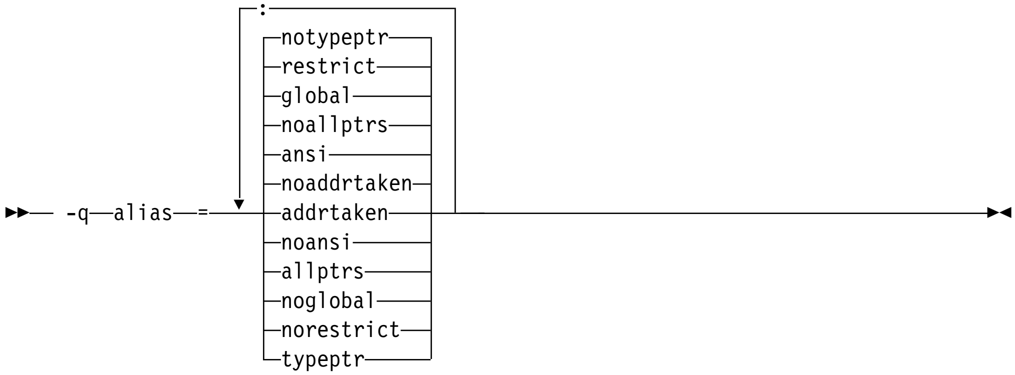
Pragma equivalent

None

Purpose

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

Syntax



Defaults

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

Parameters

addrtaken | noaddrtaken

When **addrtaken** is in effect, 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.

allptrs | noallptrs

When **allptrs** is in effect, pointers are never aliased (this also implies **-qalias=typeptr**). Specifying **allptrs** is an assertion to the compiler that no two pointers point to the same storage location. These suboptions are only valid if **ansi** is also in effect.

ansi | noansi

When **ansi** is in effect, type-based aliasing is used during optimization, which restricts the lvalues that can be safely used to access a data object. 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.

global | **noglobal**

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

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

restrict | **norestrict**

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

-qalias=restrict is independent from other **-qalias** suboptions. Using the **-qalias=restrict** option 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. You can use **norestrict** to preserve compatibility with code compiled with versions of the compiler previous to V9.0.

typeptr | **notypeptr**

When **typeptr** is in effect, pointers to different types are never aliased. The **typeptr** suboption is valid only when **ansi** is also in effect. **typeptr** is more restrictive than **ansi**. When **typeptr** is in effect, pointers can only point to an object of the same type or a compatible type, and a `char*` dereference cannot alias any other types.

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`.

The **-qalias=[no]ansi** option replaces the deprecated **-q[no]ansialias** option. Use **-qalias=[no]ansi** in your new applications.

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 193
- -qinfo=als
- "#pragma disjoint" on page 345
- *Type-based aliasing* in the *XL C Language Reference*
- *The may_alias type attribute (IBM extension)* in the *XL C Language Reference*
- *The restrict type qualifier* in the *XL C Language Reference*
- "-qrestrict" on page 266

-qalign

Category

Portability and migration

Pragma equivalent

#pragma options align, #pragma align

Purpose

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

Syntax

```
                  ┌─=power──────┐
                  ├─=full───────┤
►►── -q──align────┼─=bit_packed─┼──────────────────────────────────────────────►◄
                  ├─=mac68k─────┤
                  ├─=natural────┤
                  ├─=packed─────┤
                  └─=twobyte────┘
```

```
                         ┌─power──────┐
                         ├─full───────┤
►►──#──pragma──align──(──┼─bit_packed─┼──)─────────────────────────────────────►◄
                         ├─mac68k─────┤
                         ├─natural────┤
                         ├─packed─────┤
                         ├─twobyte────┤
                         └─reset──────┘
```

Defaults

-qalign=power

Parameters

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

`power`
Uses the RISC System/6000 alignment rules. This is the default.

`full`
Uses the RISC System/6000 alignment rules.

Note: -qalign=full is equivalent to **-qalign=power**.

`mac68k | twobyte`
Uses the Macintosh alignment rules. Valid only for 32-bit compilations.

`natural`
Structure members are mapped to their natural boundaries. This has the same effect as the **power** suboption, except that it also applies alignment rules to `double` and `long double` members that are not the first member of a structure or union.

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

Usage

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

The **full** suboption is the default to ensure compatibility with existing objects. If compatibility with earlier versions is not necessary, you should consider using **natural** alignment to improve potential application performance.

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

Note: When using **-qalign**, all system headers are also compiled with **-qalign**. For a complete explanation of the option and pragma parameters, as well as usage considerations, see "Aligning data" in the *XL C Optimization and Programming Guide*.

Predefined macros

None.

Examples

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

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

typedef struct A A2;

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

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

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

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

struct A {
int a;
#pragma options align=power /* Applies to B; A is unaffected  */
   struct B {
   char c;
   double d;
            } BB;   /*  BB uses power alignment rules  */
} AA;               /*  AA uses bit_packed alignment rules  /*
```

Related information

- "#pragma pack" on page 366
- "Aligning data" in the *XL C Optimization and Programming Guide*
- "The __align type qualifier" in the *XL C Language Reference*
- "The aligned variable attribute" in the *XL C Language Reference*
- "The packed variable attribute" in the *XL C Language Reference*

-qalloca, -ma

Category

Object code control

Pragma equivalent

#pragma alloca

Purpose

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

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

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

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

Syntax

Option syntax

```
►►─┬─ -q─alloca─┬──────────────────────────────────────────────►◄
   └─ -ma───────┘
```

Pragma syntax

```
►►─#─pragma─alloca─────────────────────────────────────────────►◄
```

Defaults

Not applicable.

Usage

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

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

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

Predefined macros

None.

Examples

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

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

Related information

- "-D" on page 126
- "__alignx" on page 607

-qaltivec

Category

Language element control

Pragma equivalent

None.

Purpose

Enables the compiler support for vector data types and operators.

Syntax

```
          ┌─noaltivec─┐
►►── -q───┴─altivec───┴──────────────────────────────────────────►◄
```

Defaults

-qnoaltivec

Usage

The `-qaltivec` option has effect only when you set or imply `-qarch` to be an architecture that supports vector instructions. Otherwise, the compiler ignores `-qaltivec` and issues a warning message.

Predefined macros

__ALTIVEC__ is defined to 1 and __VEC__ is defined to 10206 when `-qaltivec` is in effect; otherwise, they are undefined.

Related information

- "-qarch"
- "-qsimd" on page 278
- "-qvecnvol" on page 320
- *AltiVec Technology Programming Interface Manual*, available at http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf

-qarch

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

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

Syntax

```
                  ┌─pwr4───────┐
►►── -q──arch──=──┼─auto───────┼────────────────────────────────────────────►◄
                  ├─pwr5───────┤
                  ├─pwr5x──────┤
                  ├─pwr6───────┤
                  ├─pwr6e──────┤
                  ├─pwr7───────┤
                  ├─pwr8───────┤
                  ├─ppc────────┤
                  ├─ppc64v─────┤
                  ├─ppc64──────┤
                  ├─ppcgr──────┤
                  ├─ppc64gr────┤
                  ├─ppc64grsq──┤
                  └─ppc970─────┘
```

Defaults

- **-qarch=pwr4**
- **-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.

pwr4

Produces object code containing instructions that will run on the POWER4, POWER5, POWER5+, POWER6®, POWER7®, POWER7+™, POWER8®, or PowerPC® 970 hardware platforms.

pwr5

Produces object code containing instructions that will run on the POWER5, POWER5+, POWER6, POWER7, POWER7+, or POWER8 hardware platforms.

pwr5x

Produces object code containing instructions that will run on the POWER5+, POWER6, POWER7, POWER7+, or POWER8 hardware platforms.

pwr6

Produces object code containing instructions that will run on the POWER6, POWER7, POWER7+, or POWER8 hardware platforms running in POWER6, POWER7, POWER7+, or POWER8 architected mode. If you would like support for decimal floating-point instructions, be sure to specify this suboption during compilation.

pwr6e

Produces object code containing instructions that will run on the POWER6 hardware platforms running in POWER6 enhanced mode.

pwr7

Produces object code containing instructions that will run on the POWER7, POWER7+, or POWER8 hardware platforms.

pwr8

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

ppc
This suboption is deprecated. Even though it is still accepted, it is silently upgraded to **-qarch=pwr4**.

ppc64
This suboption is deprecated. Even though it is still accepted, it is silently upgraded to **-qarch=pwr4**.

ppcgr
This suboption is deprecated. Even though it is still accepted, it is silently upgraded to **-qarch=pwr4**.

ppc64gr
This suboption is deprecated. Even though it is still accepted, it is silently upgraded to **-qarch=pwr4**.

ppc64grsq
This suboption is deprecated. Even though it is still accepted, it is silently upgraded to **-qarch=pwr4**.

ppc64v
Generates instructions for generic PowerPC chips with vector processors, such as the PowerPC 970. Valid in 32-bit or 64-bit mode.

ppc970
Generates instructions specific to the PowerPC 970 architecture.

Usage

All PowerPC machines share a common set of instructions, but may also include additional instructions unique to a given processor or processor family. Using the **-qarch** option to target a specific architecture for the compilation results in code that may not run on other architectures, but provides the best performance for the selected architecture. If you want maximum performance on a specific architecture and will not be using the program on other architectures, use the appropriate architecture option. If you want to generate code that can run on more than one architecture, specify a **-qarch** suboption that supports a group of architectures. Table 22 shows the features supported by the different processor architectures and their representative **-qarch** suboptions:

Table 22. Feature support in processor architectures

Architecture	Graphics support	Square root support	64-bit support	Vector processing support	Large page support
pwr4	yes	yes	yes	no	yes
pwr5	yes	yes	yes	no	yes
pwr5x	yes	yes	yes	no	yes
ppc	yes	yes	yes	no	yes
ppc64	yes	yes	yes	no	yes
ppc64gr	yes	yes	yes	no	yes
ppc64grsq	yes	yes	yes	no	yes
ppc64v	yes	yes	yes	VMX	yes
ppc970	yes	yes	yes	VMX	yes
pwr6	yes	yes	yes	VMX	yes
pwr6e	yes	yes	yes	VMX	yes
pwr7	yes	yes	yes	VMX, VSX	yes
pwr8	yes	yes	yes	VMX, VSX	yes

Note: Vector Multimedia Extension (VMX) and Vector Scalar Extension (VSX) are processor instructions for vector processing.

For any given **-qarch** setting, the compiler defaults to a specific, matching **-qtune** setting, which can provide additional performance improvements. Alternatively, if you specify **-qarch** with a group argument, you can specify **-qtune** as either **auto** or provide a specific architecture in the group. For detailed information on using **-qarch** and **-qtune** together, see "-qtune" on page 310.

For a given application program, make sure that you specify the same **-qarch** setting when you compile each of its source files. Although the linker and loader may detect object files that are compiled with incompatible **-qarch** settings, you should not rely on it.

Predefined macros

See "Macros related to architecture settings" on page 407 for a list of macros that are predefined by **-qarch** suboptions.

Examples

To specify that the executable program `testing` compiled from `myprogram.c` is to run on a computer with VSX instruction support, enter:

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

Related information

- -qprefetch
- -qfloat
- "-qtune" on page 310
- "Specifying compiler options for architecture-specific compilation" on page 9
- "-q32, -q64" on page 94
- "Macros related to architecture settings" on page 407
- "Optimizing your applications" in the *XL C Optimization and Programming Guide*

-qasm

Category

Language element control

Pragma equivalent

None.

Purpose

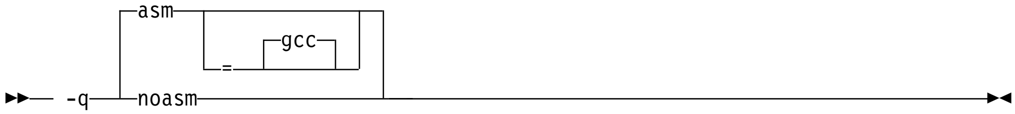
Controls the interpretation and subsequent generation of code for assembler language extensions.

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

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

Syntax

-qasm syntax (for C)



Defaults

- -qasm=gcc

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

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

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

Predefined macros

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

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 107
- “-qkeyword” on page 203
- “-qlanglvl” on page 206
- "Inline assembly statements" in the *XL C Language Reference*

-qasm_as

Category

Compiler customization

Pragma equivalent

None.

Purpose

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

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

Syntax

```
►►── -q───asm_as──=──┬─path─────────────────┬──────────────────────────────►◄
                     └─"──path──┬───────┬──"─┘
                                └─flags─┘
```

Defaults

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

Parameters

path
  The full path name of the assembler to be used.

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

Predefined macros

None.

Examples

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

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

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

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

Related information

- “-qasm” on page 105

-qassert

Category

Optimization and tuning

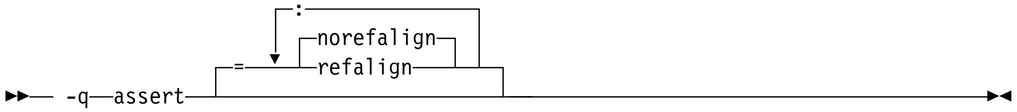
Pragma equivalent

None

Purpose

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

Syntax



Defaults

-qassert=norefalign

Parameters

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

-qattr

Category

Listings, messages, and compiler information

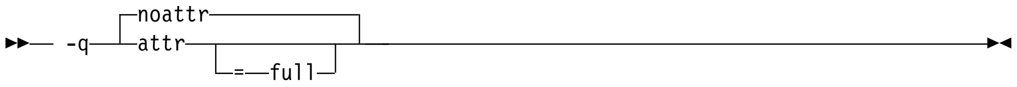
Pragma equivalent

#pragma options [no]attr

Purpose

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

Syntax



Defaults

-qnoattr

Parameters

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

Usage

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

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

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

Note: Specifying **-qattr** does not list the `#define` directives. You can use "-qshowmacros" on page 276 instead.

Predefined macros

None.

Examples

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

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

Related information

- "-qshowmacros" on page 276
- "-qprint" on page 259
- "-qxref" on page 330

-b

Category

Linking

Pragma equivalent

None.

Purpose

Sets special linker processing options. This option can be repeated.

Syntax

```
         ┌─dynamic─┐
►►── -b──┼─shared──┼──────────────────────────────────────────────────────►◄
         └─static──┘
```

Defaults

-bdynamic

Parameters

dynamic | **shared**
Causes the linker to process subsequent shared objects in dynamic mode. In dynamic mode, shared objects are not statically included in the output file. Instead, the shared objects are listed in the loader section of the output file.

static
Causes the linker to process subsequent shared objects in static mode. In static mode, shared objects are statically linked in the output file.

Usage

The default option, **-bdynamic**, ensures that the C library (libc) links dynamically. To avoid possible problems with unresolved linker errors when linking the C library, you must add the **-bdynamic** option to the end of any compilation sections that use the **-bstatic** option.

Predefined macros

Not applicable.

Related information

- "-brtl" on page 113

-B

Category

Compiler customization

Pragma equivalent

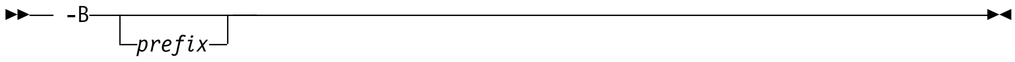
None.

Purpose

Specifies substitute path names for XL 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 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 303 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 245
- "-t" on page 303
- "Invoking the compiler" on page 1

-qbitfields

Category

Floating-point and integer control

Pragma equivalent

None.

Purpose

Specifies whether bit fields are signed or unsigned.

Syntax

```
                       ┌─unsigned─┐
►►── -q──bitfields──=──┴─signed───┴──────────────────────────────────────►◄
```

Defaults

-qbitfields=unsigned

Parameters

signed
Bit fields are signed.

unsigned
Bit fields are unsigned.

Predefined macros

None.

-bmaxdata

Category

Linking

Pragma equivalent

None

Purpose

Sets the maximum size of the area shared by the static data (both initialized and uninitialized) and the heap.

Syntax

```
►►── -bmaxdata──:────number──────────────────────────────────────────────►◄
```

Defaults

-bmaxdata:0

Parameters

number
The number of bytes used representing the soft **ulimit** set by the system loader.

- For 32-bit programs, the maximum value allowed by the system is 0x80000000 for programs that are running under large program support and 0xD0000000 for programs that are running under very large program support. For details, see Large program support in the documentation of AIX operating systems.

- For 64-bit programs, the **-bmaxdata** option provides a guaranteed maximum size for the programs data heap. You can specify any value, but the data area cannot extend past 0x06FFFFFFFFFFFFF8 regardless of the value that you specified.

Predefined macros

None.

-brtl

Category

Linking

Pragma equivalent

None.

Purpose

Enables runtime linking for the output file. When you use **-brtl** with the **-l** option, the linker searches for a library with the suffix of *.so*, as well as of *.a*. Preference is given to *.so* over *.a* when libraries with the same name are present in the same directory.

Runtime linking is the ability to resolve undefined and non-deferred symbols in shared modules after the program execution has already begun. It is a mechanism for providing runtime definitions (these function definitions are not available at link-time) and symbol rebinding capabilities. Compiling with **-brtl** adds a reference to the runtime linker to your program, which will be called by your program's startup code (/lib/crt0.o) when program execution begins. Shared object input files are listed as dependents in the program loader section in the same order as they are specified on the command line. When the program execution begins, the system loader loads these shared objects so their definitions are available to the runtime linker.

Syntax

```
►►── -brtl ─────────────────────────────────────────────────────────►◄
```

Usage

The main application must be built to enable runtime linking. The system loader must be able to load and resolve all symbols referenced in the main program and called modules, or the program will not execute. For how to link a library to an application with runtime linking enabled, see "Linking a library to an application" in the *XL C Optimization and Programming Guide*.

DCE thread libraries and heap debug libraries are not compatible with runtime linking. Do not specify the **-brtl** compiler option if you are invoking the compiler with **xlC_r4**, or if the **-qheapdebug** compiler option is specified.

Predefined macros

None.

Related information

- "-b" on page 109
- "-G" on page 163

-c

Category

Output control

Pragma equivalent

None.

Purpose

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

Syntax

```
►►— -c———————————————————————————————————————————————————————►◄
```

Defaults

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

Usage

When this option is in effect, the compiler creates an output object file, *file_name*.o, for each valid source file, such as *file_name*.c, *file_name*.i, 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 **-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 136
- "-o" on page 235
- "-P" on page 244
- "-qsyntaxonly" on page 302

-C, -C!

Category

Output control

Pragma equivalent

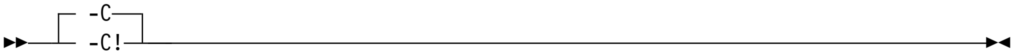
None.

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 136
- "-P" on page 244

-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

```
                   ┌─:──────────────────────┐   ┌─:─────────────────────────┐
                   ▼                        │   ▼                           │
►►─ -q─cache─=─────┬─level─=──┬─1─┬─┬───────┴───┬─assoc─=───number─┬──┴──────────►◄
                   │          ├─2─┤ │           ├─auto─────────────┤
                   │          └─3─┘ │           ├─cost─=─cycles────┤
                   └─type─=──┬─c─┬──┘           ├─line─=─bytes─────┤
                             ├─d─┤              └─size─=─Kbytes────┘
                             └─i─┘
```

Defaults

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

Parameters

`assoc`
Specifies the set associativity of the cache.

`number`
Is one of:

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

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

`cost`
Specifies the performance penalty resulting from a cache miss.

`cycles`

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

level
Is one of:

1 Basic cache

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

3 TLB

line
Specifies the line size of the cache.

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

size
Specifies the total size of the cache.

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

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

cache_type
Is one of:

c Combined data and instruction cache

d Data cache

i Instruction cache

Usage

The **-qtune** setting determines the optimal default **-qcache** settings for most typical compilations. You can use the **-qcache** to override these default settings. However, if you specify the wrong values for the cache configuration, or run the program on a machine with a different configuration, the program will work correctly but may be slightly slower.

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 116
- “-O, -qoptimize” on page 236
- “-qtune” on page 310
- “-qipa” on page 193
- "Optimizing your applications" in the *XL C Optimization and Programming Guide*

-qchars

Category

Floating-point and integer control

Pragma equivalent

#pragma options chars, #pragma chars

None.

Purpose

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

Syntax

```
                  ┌─unsigned─┐
►►── -q──chars──=──┴─signed───┴──────────────────────────────►◄
```

Pragma syntax

```
                          ┌─unsigned─┐
►►──#──pragma──chars──(──┴─signed───┴──)─────────────────────►◄
```

Defaults

-qchars=unsigned

Parameters

unsigned
Variables of type `char` are treated as `unsigned char`.

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

Usage

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

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

Predefined macros

- _CHAR_SIGNED and __CHAR_SIGNED__ are defined to 1 when **signed** is in effect; otherwise, it is undefined.
- _CHAR_UNSIGNED and __CHAR_UNSIGNED__ are defined to 1 when **unsigned** is in effect; otherwise, they are undefined.

Examples

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

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

-qcheck

Category

Error checking and debugging

Pragma equivalent

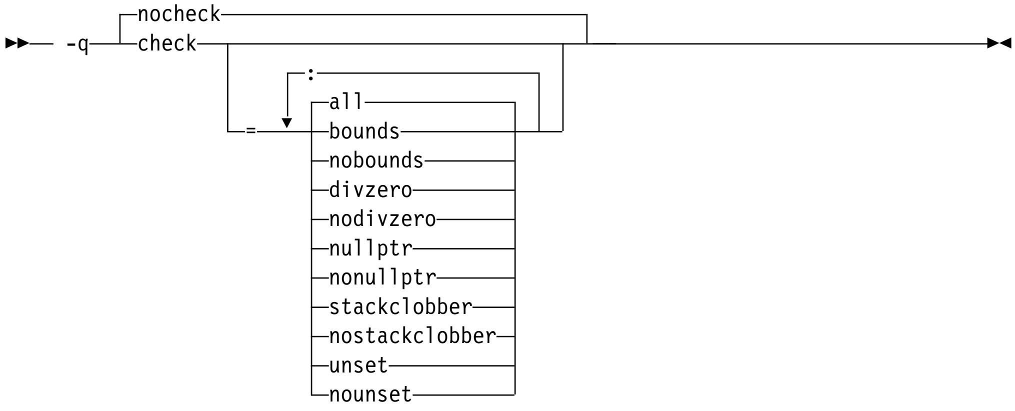
#pragma options [no]check

Purpose

Generates code that performs certain types of runtime checking.

If a violation is encountered, a runtime error is raised by sending a SIGTRAP signal to the process. 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.

  If the **-qstackprotect** option and this suboption are both on, this suboption detects the stack corruption first.

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

Pragma equivalent

#pragma options [no]compact

Purpose

Avoids optimizations that increase code size.

Syntax

```
          ┌─nocompact─┐
►►── -q───┴─compact───┴──────────────────────────────────────────────►◄
```

Defaults

-qnocompact

Usage

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

This option takes effect only when it is specified at the **-O2** optimization level, or higher.

Predefined macros

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

Examples

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

```
xlc myprogram.c -O -qcompact
```

-qconcurrentupdate

Category

Object code control

Pragma equivalent

None.

Purpose

Supports updating the operating system while the kernel is running.

Syntax

```
          ┌─noconcurrentupdate─┐
►►── -q───┴─concurrentupdate───┴──────────────────────────────────────────►◄
```

Defaults

-qnoconcurrentupdate

Usage

If you want to use AIX Concurrent Update (hot-patch), you must use **-qconcurrentupdate** to compile your code. For details about Concurrent Update, see the AIX Concurrent Update documentation.

Predefined macros

None.

Examples

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

-qcpluscmt

Category

Language element control

Pragma equivalent

None.

Purpose

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

Syntax

```
          ┌─cpluscmt───┐
►►── -q───┴─nocpluscmt─┴──────────────────────────────────────────────────►◄
```

Defaults

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

Predefined macros

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

Examples

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

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

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

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

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

Related information

- "-C, -C!" on page 115
- "-qlanglvl" on page 206
- "Comments" in the *XL C Language Reference*

-qcrt

Category

Linking

Pragma equivalent

None.

Purpose

Specifies whether system startup files are to be linked.

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

```
         ┌─crt───┐
►►─ -q───┴─nocrt─┴──────────────────────────────────────────────────►◄
```

Defaults

-qcrt

Predefined macros

None.

Related information

- "-qlib" on page 213

-qc_stdinc

Category

Compiler customization

Pragma equivalent

None.

Purpose

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

Syntax

```
                               ┌─ : ──────────────┐
                               ▼                  │
►►─ -q─c_stdinc─=─┬─────┬────directory_path───────┴─┬─────┬──────────►◄
                  └─"───┘                           └─"───┘
```

Defaults

By default, the compiler searches the directories specified in the configuration file for the XL C header files (this is normally /opt/IBM/xlc/13.1.3/include/) and for the system header files (this is normally /usr/include/).

Parameters

directory_path
The path for the directory where the compiler should search for the XL C and

system 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 and system headers, you use a configuration file to do so; see "Directory search sequence for included files" on page 12 for more information.

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

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

Predefined macros

None.

Examples

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

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

Related information

- "-qstdinc" on page 292
- "-qinclude" on page 176
- "Directory search sequence for included files" on page 12
- "Specifying compiler options in a configuration file" on page 7

-D

Category

Language element control

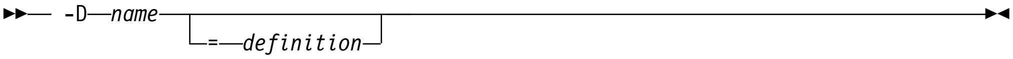
Pragma equivalent

None.

Purpose

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

Syntax



Defaults

Not applicable.

Parameters

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

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

Usage

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

To aid in program portability and standards compliance, the operating system provides several header files that refer to macro names you can set with the **-D** option. You can find most of these header files either in the /usr/include directory or in the /usr/include/sys directory. To ensure that the correct macros for your source file are defined, use the **-D** option with the appropriate macro name. For example, if your source file includes the /usr/include/sys/stat.h header file, you must compile with the option -D_POSIX_SOURCE to pick up the correct definitions for that file.

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

AIX 4.2 and later provides support for files greater than 2 gigabytes in size so you can store large quantities of data in a single file. To allow large file manipulation in your application, compile with the -D_LARGE_FILES and **-qlonglong** compiler options. For example:

```
xlc myprogram.c -D_LARGE_FILES -qlonglong
```

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 313
- Chapter 6, "Compiler predefined macros," on page 403
- "Header files" in the *AIX Files Reference*

-qdataimported, -qdatalocal, -qtocdata

Category

Optimization and tuning

Pragma equivalent

None.

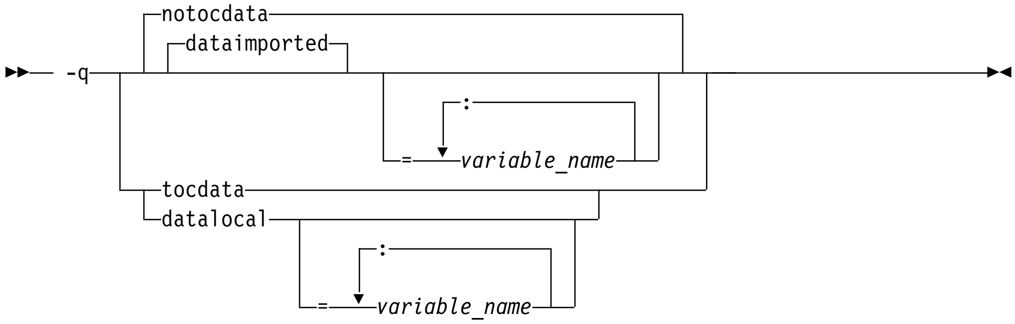
Purpose

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).

  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, performance may decrease.

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

Predefined macros

None.

Related information

- "-qprocimported, -qproclocal, -qprocunknown" on page 260

-qdbgfmt

Category

Error checking and debugging

Pragma equivalent

None.

Purpose

Specifies the format for the debugging information in object files.

DWARF is a standard that defines the format of debugging information in programs. It is used on a wide variety of operating systems and is extensible and compact.

Syntax

```
                       ┌─stabstring─┐
►►── -q──dbgfmt──=─────┼─dwarf──────┼──────────────────────────────────────────────►◄
                       └─dwarf4─────┘
```

Defaults

-qdbgfmt=stabstring

Parameters

stabstring
Generates debugging information in stabstring format.

C11 **Note:** This suboption does not generate debugging information for C11 features. Use the **dwarf** or **dwarf4** suboption instead for these features. C11

dwarf
Generates debugging information in DWARF 3 format.

dwarf4
Generates debugging information in DWARF 4 format.

Notes:

- To use **-qdbgfmt=dwarf** or **-qdbgfmt=dwarf4**, the program must be compiled and linked on AIX V7.1 or above.
- To debug programs built with **-qdbgfmt=dwarf** or **-qdbgfmt=dwarf4**, a DWARF-enabled debugger such as **dbx** is required.

Usage

-qdbgfmt does not imply any of the debugging options, such as **"-g" on page 160**. To generate debugging information, you must specify a debugging option, for example:

- To generate debugging information in stabstring format, use **-g -qdbgfmt=stabstring**.
- To generate debugging information in DWARF 3 format, use **-g -qdbgfmt=dwarf**.
- To generate debugging information in DWARF 4 format, use **-g -qdbgfmt=dwarf4**.

-qdbgfmt also applies to **"-qlinedebug" on page 216**, which generates a subset of **"-g" on page 160** information. For example, you can use **-qlinedebug -qdbgfmt=dwarf** to generate line number information in DWARF 3 format.

Related information

- "-g" on page 160
- "-qlinedebug" on page 216

-qdbxextra

Category

Error checking and debugging

Pragma equivalent

#pragma options dbxextra

Purpose

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

To minimize the size of object and executable files, the compiler only includes information for `typedef` declarations, `struct`, `union`, and `enum` type definitions that are referenced by the program. When you specify the **-qdbxextra** option, debugging information is included in the symbol table of the object file. This option is equivalent to the **-qsymtab=unref** option.

Syntax

```
            ┌─nodbxextra─┐
►►── -q─────┴─dbxextra───┴──────────────────────────────────────────►◄
```

Defaults

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

Usage

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

Predefined macros

None.

Examples

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

```
xlc myprogram.c -g -qdbxextra
```

Related information

- "-qfullpath" on page 156
- "-qlinedebug" on page 216
- "-g" on page 160
- "#pragma options" on page 362
- "-qsymtab" on page 301

-qdfp

Category

Language element control

Pragma equivalent

None.

Purpose

Enables compiler support for decimal floating-point types and literals.

Syntax

```
          ┌─nodfp─┐
►►──-q────┴─dfp───┴──────────────────────────────────────────────────►◄
```

Defaults

-qnodfp

Usage

If you enable **-qdfp** for a **-qarch** value that does not support decimal floating-point instructions, **-qfloat=dfpemulate** is automatically enabled, and the decimal floating-point operations are performed by software. This may cause a slowdown in the application's runtime performance.

Note: To use decimal floating-point types and literals, you must also enable specific code in header files by defining the __STDC_WANT_DEC_FP__ macro at compile time. See "Examples" on page 132.

Predefined macros

When **-qdfp** is in effect, __IBM_DFP__ is predefined to a value of 1; otherwise it is undefined.

Examples

To compile `myprogram.c` that contains decimal floating-point type and literal, enter:

```
xlc myprogram.c -qarch=pwr7 -qdfp -D__STDC_WANT_DEC_FP__
```

Related information

- Compiling a decimal floating-point program
- “-qarch” on page 102
- “-qfloat” on page 146
- “-D” on page 126

-qdigraph

Category

Language element control

Pragma equivalent

#pragma options [no]digraph

Purpose

Enables recognition of digraph key combinations to represent characters that are not found on some keyboards. Digraph key combinations include `<:`, `<%`, and so on.

Syntax

```
          ┌─digraph───┐
►►── -q───┴─nodigraph─┴──────────────────────────────────────────────►◄
```

Defaults

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

Usage

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

Predefined macros

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

Examples

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

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

Related information

- "-qlanglvl" on page 206
- "-qtrigraph" on page 310

-qdirectstorage

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

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

Syntax

```
            ┌─nodirectstorage─┐
►►── -q─────┴─directstorage───┴─────────────────────────────────────────►◄
```

Defaults

-qnodirectstorage

Usage

Use this option with discretion. It is intended for programmers who know how the memory and cache blocks work, and how to tune their applications for optimal performance. To ensure that your application will execute correctly on all implementations, you should assume that separate instruction and data caches exist and program your application accordingly.

-qdollar

Category

Language element control

Pragma equivalent

#pragma options [no]dollar

Purpose

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

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

Syntax

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

Defaults

-qdollar

Usage

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

Predefined macros

None.

Examples

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

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

Related information

- "-qlanglvl" on page 206

-qdpcl

Category

Error checking and debugging

Pragma equivalent

None.

Purpose

Generates symbols that tools based on the IBM Dynamic Probe Class Library (DPCL) can use to see the structure of an executable file.

DPCL is an open-source set of libraries used by application performance analysis tools (for more information, visit http://dpcl.sourceforge.net). When **-qdpcl** is in effect, the compiler emits symbols to define blocks of code in a program; you can then use tools that use the DPCL interface to examine performance information such as memory usage for object files compiled with this option.

Syntax

```
           ┌─nodpcl─┐
►►── -q────┴─dpcl───┴────────────────────────────────────────────────►◄
```

Defaults

-qnodpcl

Usage

You must specify **-qdpcl** together with the **-g** option to ensure that the compiler generates debugging information required by debugging and program analysis tools.

-qdpcl is not supported for any optimization level except zero. If a non-zero optimization level is specified or implied by other options, **-qdpcl** will be disabled.

You cannot specify the **-qipa** or **-qsmp** options together with **-qdpcl**.

Predefined macros

None.

Related information

- “-g” on page 160
- “-qipa” on page 193
- “-qsmp” on page 281

-e

Category

Linking

Pragma equivalent

None.

Purpose

When used together with the **-qmkshrobj** option or **-G** option, specifies an entry point for a shared object.

Syntax

```
▸▸── -e───entry_name──────────────────────────────────────────────▸◂
```

Defaults

Not applicable.

Parameters

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

Usage

Specify the **-e** option only with the **-qmkshrobj** or **-G** 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

- "-qmkshrobj" on page 233
- "-G" on page 163

-E

Category

Output control

Pragma equivalent

None.

Purpose

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

Syntax

```
►►— -E——————————————————————————————————————————————————————►◄
```

Defaults

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

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

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

Predefined macros

None.

Examples

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

```
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:

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

int c ;

c = a + b ;
```

Related information

- "-qppline" on page 255
- "-C, -C!" on page 115
- "-P" on page 244
- "-qsyntaxonly" on page 302

-qenum

Category

Floating-point and integer control

Pragma equivalent

#pragma options enum, #pragma enum

Purpose

Specifies the amount of storage occupied by enumerations.

Syntax

Option syntax

```
                  ┌─intlong─┐
►►── -q──enum──=──┼─int─────┼──────────────────────────────────►◄
                  ├─small───┤
                  ├─1───────┤
                  ├─2───────┤
                  ├─4───────┤
                  └─8───────┘
```

Pragma syntax

```
                          ┌─intlong─┐
►►──#──pragma──enum────(──┼─int─────┼──)─────────────────────────────────────►◄
                          ├─small───┤
                          ├─1───────┤
                          ├─2───────┤
                          ├─4───────┤
                          ├─8───────┤
                          ├─pop─────┤
                          └─reset───┘
```

Defaults

-qenum=intlong

Parameters

1 Specifies that enumerations occupy 1 byte of storage, are of type `signed char` if the range of enumeration values falls within the limits of `signed char`, and `unsigned char` otherwise.

2 Specifies that enumerations occupy 2 bytes of storage, are of type `short` if the range of enumeration values falls within the limits of `signed short`, and `unsigned short` otherwise. Values cannot exceed the range of `signed int`.

4 Specifies that enumerations occupy 4 bytes of storage, are of type `int` if the range of enumeration values falls within the limits of `signed int`, and `unsigned int` otherwise.

8 Specifies that enumerations occupy 8 bytes of storage. In 32-bit compilation mode, the enumeration is of type `long long` if the range of enumeration values falls within the limits of `signed long long`, and `unsigned long long` otherwise. In 64-bit compilation mode, the enumeration is of type `long` if the range of enumeration values falls within the limits of `signed long`, and `unsigned long` otherwise.

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

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

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

pop | reset (pragma only)
Discards the current pragma setting and reverts to the setting specified by the previous pragma directive. If no previous pragma was specified, reverts to the command-line or default option setting.

Usage

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

Table 23. Enumeration sizes and types

	enum=1		enum=2		enum=4		enum=8			
							32-bit compilation mode		64-bit compilation mode	
Range	var	const	var	const	var	const	var	const	var	const
0..127	signed char	int	short	int	int	int	long long	long long	long	long
-128..127	signed char	int	short	int	int	int	long long	long long	long	long
0..255	unsigned char	int	short	int	int	int	long long	long long	long	long
0..32767	ERROR[1]	int	short	int	int	int	long long	long long	long	long
-32768..32767	ERROR[1]	int	short	int	int	int	long long	long long	long	long
0..65535	ERROR[1]	int	unsigned short	int	int	int	long long	long long	long	long
0..2147483647	ERROR[1]	int	ERROR[1]	int	int	int	long long	long long	long	long
-(2147483647+1) ..2147483647	ERROR[1]	int	ERROR[1]	int	int	int	long long	long long	long	long
0..4294967295	ERROR[1]	unsigned int[2]	ERROR[1]	unsigned int[2]	unsigned int[2]	unsigned int[2]	long long	long long	long	long
0..(2^{63}-1)	ERROR[1]	long[2]	ERROR[1]	long[2]	ERROR[1]	long[2]	long long[2]	long long[2]	long[2]	long[2]
-2^{63}..(2^{63}-1)	ERROR[1]	long[2]	ERROR[1]	long[2]	ERROR[1]	long[2]	long long[2]	long long[2]	long[2]	long[2]
0..2^{64}	ERROR[1]	unsigned long[2]	ERROR[1]	unsigned long[2]	ERROR[1]	unsigned long[2]	unsigned long long[2]	unsigned long long[2]	unsigned long[2]	unsigned long[2]

	enum=int		enum=intlong				enum=small			
			32-bit compilation mode		64-bit compilation mode		32-bit compilation mode		64-bit compilation mode	
Range	var	const	var	const	var	const	var	const	var	const
0..127	int	int	int	int	int	int	unsigned char	int	unsigned char	int
-128..127	int	int	int	int	int	int	signed char	int	signed char	int
0..255	int	int	int	int	int	int	unsigned char	int	unsigned char	int
0..32767	int	int	int	int	int	int	unsigned short	int	unsigned short	int
-32768..32767	int	int	int	int	int	int	short	int	short	int
0..65535	int	int	int	int	int	int	unsigned short	int	unsigned short	int
0..2147483647	int	int	int	int	int	int	unsigned int	unsigned int	unsigned int	unsigned int

-(2147483647+1) ..2147483647	int	int	int	int	int	int	int	int	int	int
0..4294967295	unsigned int[1]	unsigned int[2]	unsigned int[2]	unsigned int[2]	unsigned int[2]	unsigned int[2]	unsigned int[2]	unsigned int[2]	unsigned int[2]	unsigned int[2]
0..(2^{63}-1)	ERR[2]	ERR[2]	long long[2]	long long[2]	long[2]	long[2]	unsigned long long[2]	unsigned long long[2]	unsigned long[2]	unsigned long[2]
-2^{63}..(2^{63}-1)	ERR[2]	ERR[2]	long long[2]	long long[2]	long[2]	long[2]	long long[2]	long long[2]	long[2]	long[2]
0..2^{64}	ERR[2]	ERR[2]	unsigned long long[2]	unsigned long long[2]	unsigned long[2]	unsigned long[2]	unsigned long long[2]	unsigned long long[2]	unsigned long[2]	unsigned long[2]

Notes:

- These enumerations are too large for the **-qenum=1|2|4|int** setting. A Severe error is issued and compilation stops. To correct this condition, you should reduce the range of the enumerations, choose a larger **-qenum** setting, or choose a dynamic **-qenum** setting, such as **small** or **intlong**.
- Enumeration types must not exceed the range of `int` when compiling C applications to ISO C 1989 and ISO C 1999 Standards. With the **stdc89 | stdc99** language level in effect, the compiler will behave as follows if the value of an enumeration exceeds the range of `int` and the **-qenum** option in effect supports this value:
  - If **-qenum=int** is in effect, a severe error message is issued and compilation stops.
  - For all other settings of **-qenum**, an informational message is issued and compilation continues.

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

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

Examples

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

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

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

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

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

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

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

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

Predefined macros

None.

-qexpfile

Category

Object code control

Pragma equivalent

None.

Purpose

When used together with the **-qmkshrobj** or **-G** option, saves all exported symbols in a designated file.

Syntax

```
►►— -q—expfile—=—filename————————————————————————————————►◄
```

Parameters

filename
    The name of the file to which exported symbols are written.

Usage

This option is valid only when used with the **-qmkshrobj** or **-G** option.

Predefined macros

None.

Related information

- "-qmkshrobj" on page 233
- "-G" on page 163

-qextchk

Category

Error checking and debugging

Pragma equivalent

#pragma options [no]extchk

Purpose

Generates link-time type checking information and checks for compile-time consistency.

Syntax

```
          ┌─noextchk─┐
►►── -q───┴─extchk───┴─────────────────────────────────────────────►◄
```

Defaults

-qnoextchk

Usage

This option does not perform type checking on functions or objects that contain references to incomplete types.

Predefined macros

None.

Examples

To compile `myprogram.c` so that link-time checking information is produced, enter:

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

-f

Category

Linking

Pragma equivalent

None.

Purpose

Names a file that stores a list of object files for the compiler to pass to the linker.

Syntax

```
►►── -f──filelistname──────────────────────────────────────────────►◄
```

Usage

The *filelistname* file should contain only the names of object files. There should be one object file per line.

This option is the same as the **-f** option for the **ld** command.

Predefined macros

None.

Examples

To pass the list of files contained in `myobjlistfile` to the linker, enter:

```
xlc -f/usr/tmp/myobjlistfile
```

-F

Category

Compiler customization

Pragma equivalent

None.

Purpose

Names an alternative configuration file or stanza for the compiler.

Syntax

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

Defaults

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

Parameters

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

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

Usage

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

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

Predefined macros

None.

Examples

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

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

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

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

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

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

Related information

- “Using custom compiler configuration files” on page 38
- “-B” on page 110
- “-t” on page 303
- “-W” on page 326
- “Specifying compiler options in a configuration file” on page 7
- “Compile-time and link-time environment variables” on page 24

-qfdpr

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

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

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

Syntax

```
            ┌─nofdpr─┐
►►── -q─────┴─fdpr───┴─────────────────────────────────────────────────────►◄
```

Defaults

-qnofdpr

Usage

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

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

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

Predefined macros

None.

Examples

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

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

Related information

- "-qpdf1, -qpdf2" on page 247

-qflag

Category

Listings, messages, and compiler information

Pragma equivalent

#pragma options flag

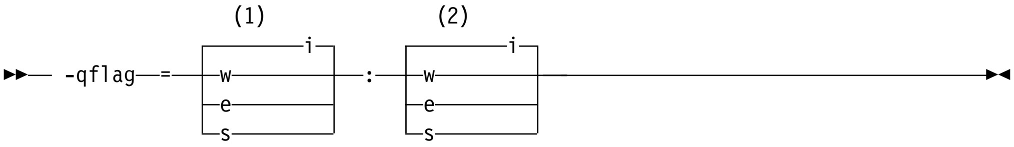
Purpose

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

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

Syntax

-qflag syntax – C



Notes:

1 Minimum severity level of messages reported in listing

2 Minimum severity level of messages reported on terminal

Defaults

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

Parameters

i Specifies that all diagnostic messages are to display: warning, error and informational messages. Informational messages (I) are of the lowest severity.

w Specifies that warning (W) and all types of error messages are to display.

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

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

Usage

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

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

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

Predefined macros

None.

Examples

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

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

Related information

- “-qinfo” on page 178
- “-qhaltonmsg” on page 166
- “-w” on page 325
- “Compiler messages” on page 16

-qfloat

Category

Floating-point and integer control

Pragma equivalent

#pragma options float

Purpose

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

Syntax

```
                 ┌─:──────────────────┐
                 │ ┌─nosubnormals─┐   │
                 │ ├─nospnans─────┤   │
                 │ ├─single───────┤   │
                 │ ├─norsqrt──────┤   │
                 │ ├─norrm────────┤   │
                 │ ├─rngchk───────┤   │
                 │ ├─rndsngl──────┤   │
                 │ ├─norelax──────┤   │
                 │ ├─nonans───────┤   │
                 │ ├─maf──────────┤   │
                 │ ├─nohssngl─────┤   │
                 │ ├─nohsflt──────┤   │
                 │ ├─nohscmplx────┤   │
                 │ ├─fold─────────┤   │
                 │ ├─nofltint─────┤   │
                 │ ├─nofenv───────┤   │
                 ▼ ├─dfpemulate───┤   │
►►── -q─float──=───┼─nodfpemulate─┼───┴──────────────────────────────────►◄
                   ├─fenv─────────┤
                   ├─fltint───────┤
                   ├─nofold───────┤
                   ├─hscmplx──────┤
                   ├─hsflt────────┤
                   ├─hssngl───────┤
                   ├─nomaf────────┤
                   ├─nans─────────┤
                   ├─relax────────┤
                   ├─norndsngl────┤
                   ├─norngchk─────┤
                   ├─rrm──────────┤
                   ├─rsqrt────────┤
                   ├─nosingle─────┤
                   ├─spnans───────┤
                   └─subnormals───┘
```

Defaults

- **-qfloat=dfpemulate:nofenv:nofltint:fold:nohscmplx:nohsflt:nohssngl:maf: nonans:norelax:rndsngl:rngchk:norrm:norsqrt:single:nospnans:nosubnormals**
- **-qfloat=fltint:rsqrt:norngchk:nosubnormals** when **-qnostrict**, **-qstrict=nooperationprecision:noexceptions**, or the **-O3** or higher optimization level is in effect.

Parameters

dfpemulate | nodfpemulate

Specifies whether decimal floating-point computations are implemented in hardware instructions or emulated in software by calls to library functions. **nodfpemulate** is only valid on a system that supports decimal floating-point instructions; that is, a system with **-qarch=pwr6** or above in effect. **nodfpemulate** is the recommended setting for those systems, and results in improved performance of decimal floating-point operations and overall program runtime performance. **dfpemulate** is required for all other **-qarch** values.

Note that **-qdfp** must also be enabled for either suboption to have any effect. Otherwise, **nodfpemulate** is set.

fenv | nofenv

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

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

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

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

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

fltint | nofltint

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

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

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

fold | nofold

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

The **-qfloat=[no]fold** option replaces the deprecated **-q[no]fold** option. Use **-qfloat=[no]fold** in your new applications.

hscmplx | nohscmplx

Speeds up operations involving complex division and complex absolute value. This suboption, which provides a subset of the optimizations of the **hsflt** suboption, is preferred for complex calculations.

hsflt | nohsflt

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

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

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

`hssngl | nohssngl`

Specifies that single-precision expressions are rounded only when the results are stored into memory locations, but not after expression evaluation. Using **hssngl** can improve runtime performance and is safer than using **hsflt**.

This option only affects double-precision (double) expressions cast to single-precision (float) and used in an assignment operator for which a store instruction is generated, when **-qfloat=nosingle** is in effect. Do not use this option if you are compiling with the default **-qfloat=single**.

`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.

The **-qfloat=[no]maf** option replaces the deprecated **-q[no]maf** option. Use **-qfloat=[no]maf** in your new applications.

`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.

The **hsflt** option overrides the **nans** option.

The **-qfloat=[no]nans** option replaces the deprecated **-qfloat=[no]spnans** option and the **-q[no]spnans** option. Use **-qfloat=[no]nans** in your new applications.

`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.

`norndsngl | rndsngl`

Rounds the result of each single-precision operation to single-precision, rather than waiting until the full expression is evaluated. It sacrifices speed for consistency with results from similar calculations on other types of computers.

This option only affects double-precision expressions cast to single-precision. You can only specify **norndsngl** when **-qfloat=nosingle** is in effect.

The **hsflt** suboption overrides the **rndsngl** option.

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.

The **-qfloat=[no]rrm** option replaces the deprecated **-q[no]rrm** option. Use **-qfloat=[no]rrm** in your new applications.

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**.

single | nosingle

Allows single-precision arithmetic instructions to be generated for single-precision floating-point values. All Power processors support single-precision instructions; however, if you want to preserve the behavior of applications compiled for earlier architectures, in which all floating-point arithmetic was performed in double-precision and then truncated to single-precision, you can use **-qfloat=nosingle:norndsngl**. This suboption provides computation precision results compatible with those provided by the deprecated options **-qarch=com | pwr | pwrx | pwr2 | p2sc | 601 | 602 | 603**. **-qfloat=nosingle** can be specified in 32-bit mode only.

`spnans | nospnans`

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

The **hsflt** suboption overrides the **spnans** suboption.

`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: This suboption takes effect only on POWER8 processors. To use this suboption, you must also specify the **-qarch=pwr8** and **-qtune=pwr8** options.

Note: For details about the relationship between **-qfloat** suboptions and their **-qstrict** counterparts, see "-qstrict" on page 294.

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

__IBM_DFP_SW_EMULATION__ is predefined to a value of 1 when **-qfloat=dfpemulate** is in effect; otherwise it is undefined.

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

- "-qarch" on page 102
- "-qflttrap"
- "-qldbl128, -qlongdouble" on page 212
- "-qstrict" on page 294
- "Handling floating-point operations" in the *XL C Optimization and Programming Guide*

-qflttrap

Category

Error checking and debugging

Pragma equivalent

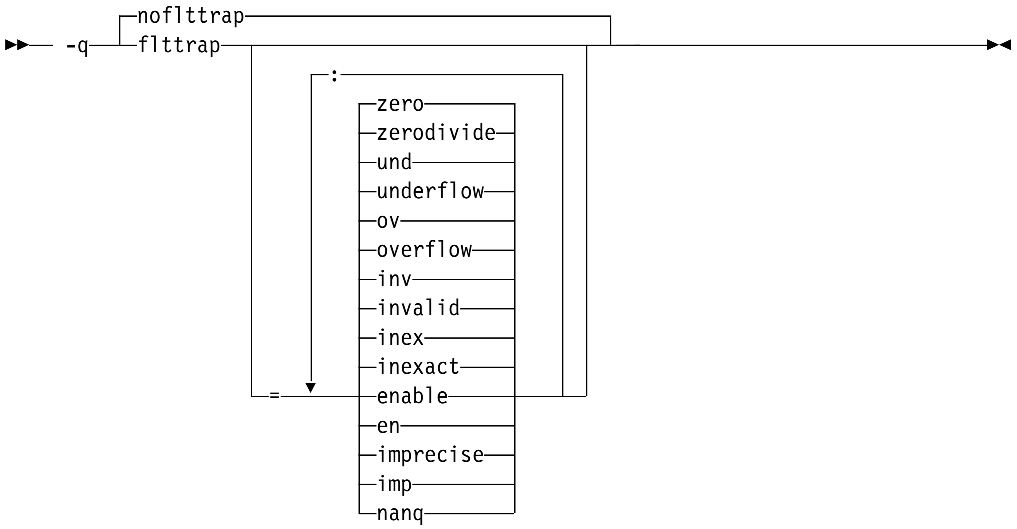
#pragma options [no]flttrap

Purpose

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

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

Syntax



Defaults

-qnoflttrap

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

Parameters

`enable, en`
Inserts a trap when the specified exceptions (**overflow**, **underflow**, **zerodivide**, **invalid**, or **inexact**) occur. You must specify this suboption if you want to turn on exception trapping without modifying your source code. If any of the specified exceptions occur, a SIGTRAP or SIGFPE signal is sent to the process with the precise location of the exception. If **imprecise** is in effect, traps will not report exactly where the exception occurred.

`imprecise, imp`
Enables imprecise detection of the specified exceptions. The compiler generates instructions after a block of code and just before the `main` program returns, to check if any of the specified exceptions (**overflow**, **underflow**, **zerodivide**, **invalid**, or **inexact**) have occurred. If an exception has occurred, an exception status flag is set in the Floating-Point Status and Control Register, but the exact location of the exception is not determined. Because instructions are not

generated after each floating-point operation and function call to check for exceptions, this suboption can result in a slight performance improvement.

inexact, **inex**
: Enables the detection of floating-point inexact operations. If **imprecise** is not also specified, the compiler generates instructions after each floating-point operation and function call to check if an inexact operation exception has occurred. 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 **imprecise** is not also specified, the compiler generates instructions after each floating-point operation and function call to check if an invalid operation exception has occurred. 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 **imprecise** is not also specified, the compiler generates instructions after each floating-point operation and function call to check if an overflow exception has occurred. 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 **imprecise** is not also specified, the compiler generates instructions after each floating-point operation and function call to check if an underflow exception has occurred. 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 **imprecise** is not also specified, the compiler generates instructions after each floating-point operation and function call to check if a zero-divide exception has occurred. 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 **-qflttrap**. This ensures that the compiler will generate the code to automatically enable floating-point exception trapping, without requiring that you include calls to the appropriate floating-point exception library functions in your code.

If you specify **-qflttrap** more than once, both with and without suboptions, the **-qflttrap** without suboptions is ignored.

The **-qflttrap** option is recognized during linking with IPA. Specifying the option at the link step overrides the compile-time setting.

If your program contains signalling NaNs, you should use the **-qfloat=nans** option along with **-qflttrap** to trap any exceptions.

The compiler exhibits behavior as illustrated in the following examples when the **-qflttrap** option is specified together with an optimization option:

- with **-O2**:
  - 1/0 generates a **div0** exception and has a result of infinity
  - 0/0 generates an invalid operation
- with **-O3** or greater:
  - 1/0 generates a **div0** exception and has a result of infinity
  - 0/0 returns zero multiplied by the result of the previous division.

If you use **-qflttrap=inv:en** to compile a program containing an IEEE invalid SQRT operation and you specify a **-qarch** target that does not implement the `sqrt` instruction set, the expected SIGTRAP signal will not occur when you run the program. You can fix this problem by specifying the following command before running the program:

```
export SQRT_EXCEPTION=3.1
```

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 -qflttrap=zerodivide:enable divide_by_zero.c
```

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

Related information

- "-qfloat" on page 146
- "-qarch" on page 102

-qformat

Category

Error checking and debugging

Pragma equivalent

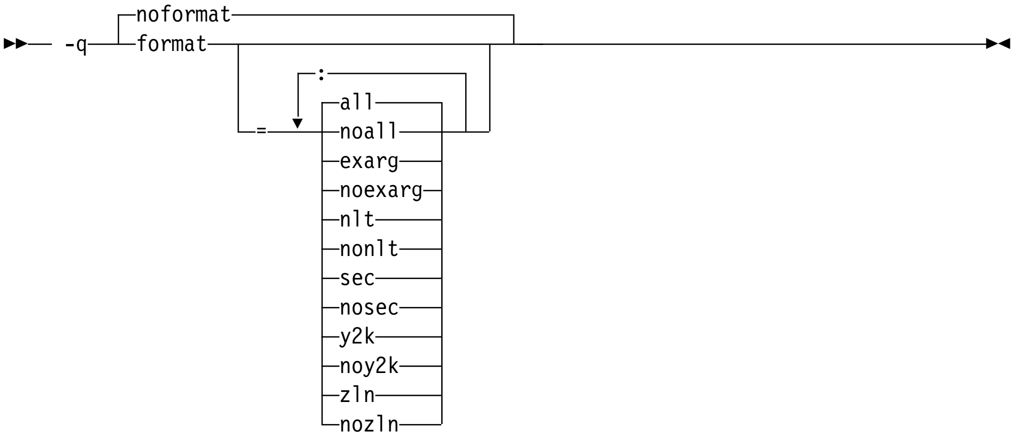
None.

Purpose

Warns of possible problems with string input and output format specifications.

Functions diagnosed are `printf`, `scanf`, `strftime`, `strfmon` family functions and functions marked with format attributes.

Syntax



Defaults

-qnoformat

Parameters

all | noall
Enables or disables all format diagnostic messages.

exarg | noexarg
Warns if excess arguments appear in `printf` and `scanf` style function calls.

nlt | nonlt
Warns if a format string is not a string literal, unless the format function takes its format arguments as a `va_list`.

sec | nosec
Warns of possible security problems in use of format functions.

y2k | noy2k
Warns of `strftime` formats that produce a 2-digit year.

zln | nozln
Warns of zero-length formats.

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

-qnoformat is equivalent to **-qformat=noall**.

Predefined macros

None.

Examples

To enable all format string diagnostics, enter either of the following:

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

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

```
xlc myprogram.c -qformat=all:noy2k
```

-qfullpath

Category

Error checking and debugging

Pragma equivalent

#pragma options [no]fullpath

Purpose

When used with the **-g** or **-qlinedebug** option, this option records the full, or absolute, path names of source and include files in object files compiled with debugging information, so that debugging tools can correctly locate the source files.

When **fullpath** is in effect, the absolute (full) path names of source files are preserved. When **nofullpath** is in effect, the relative path names of source files are preserved.

Syntax

```
          ┌─nofullpath─┐
►►── -q───┴─fullpath───┴─────────────────────────────────────────►◄
```

Defaults

-qnofullpath

Usage

If your executable file was moved to another directory, the debugger would be unable to find the file unless you provide a search path in the debugger. You can use **fullpath** to ensure that the debugger locates the file successfully.

Predefined macros

None.

Related information

- “-qlinedebug” on page 216
- “-g” on page 160

-qfuncsect

Category

Object code control

Pragma equivalent

#pragma options [no]funcsect

Purpose

Places instructions for each function in a separate object file control section or CSECT. Placing each function in its own section or CSECT might reduce the size of your program because the linker can collect garbage per function rather than per object file.

When **-qfuncsect** is specified, the compiler generates references from each function to the static data area, if one exists, in order to ensure that if any function from that object file is included in the final executable, the static data area also is included. This is done to ensure that any static strings or strings from a pragma comment, possible containing copyright information, are also included in the executable. This can, in some cases, cause code bloat or unresolved symbols at link time.

When **-qnofuncsect** is in effect, each object file consists of a single control section combining all functions defined in the corresponding source file. You can use **-qfuncsect** to place each function in a separate control section.

Syntax

```
         ┌─nofuncsect────────────────────────────┐
►►── -q──┴─funcsect──┬───────────────────────────┬┴────────────────────────►◄
                     │      ┌─implicitstaticref───┐│
                     └─=────┴─noimplicitstaticref─┴┘
```

Defaults

-qnofuncsect

Parameters

implicitstaticref | noimplicitstaticref

Specifies whether references to the static data section of the object file by functions that are contained in static variables, virtual function tables, or exception handling tables, are maintained.

In releases before XL C for AIX V11.1, all exception handling tables were placed in one static data section. Including one exception handling table meant all the other tables were also included. Therefore, references to functions in the unused exception handling tables prevented linker garbage collection of those functions, which would otherwise have been cleaned up. Starting from XL C for AIX, V11.1, this problem is solved by allocating each exception handling table its own TOC entry. As a result, the size of the final executable might be reduced.

Note: The XL C for AIX, V11.1 enhancement enables large TOC access, which sets no limit on the number of TOC entries.

When your code contains a **#pragma comment** directive or a static string for copyright information purposes, the compiler automatically places these strings in the static data area and generates references to these static data areas in the object code.

When **-qfuncsect=implicitstaticref** is in effect, a reference to the static area is generated even if not otherwise referenced.

When **-qfuncsect=noimplicitstaticref** is in effect, a reference to the static area is only generated if referenced by the program.

Specifying **-qfuncsect** with no suboption implies **-qfuncsect=implicitstaticref**.

Usage

Using multiple control 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.

Related information

- “#pragma comment” on page 343

-qfunctrace

Category

Error checking and debugging

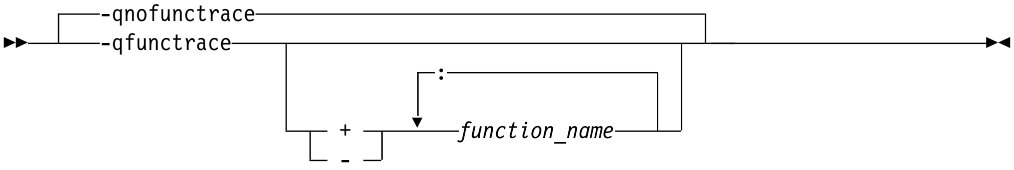
Pragma equivalent

None.

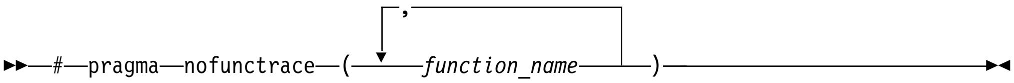
Purpose

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

Syntax



Pragma syntax



Defaults

-qnofunctrace

Parameters

+ Instructs the compiler to trace *function_name* and all its internal functions.

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

`function_name`
Indicates the named functions to be traced.

Usage

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

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

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

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

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

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

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

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

Note:

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

Predefined macros

None.

Examples

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

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

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

- `-qfunctrace+x -qfunctrace+y` or `-qfunctrace+x -qnofunctrace -qfunctrace+y` enables tracing for only `x` and `y`.
- `-qfunctrace-x -qfunctrace` or `-qfunctrace -qfunctrace-x` traces all functions in the compilation unit except for `x`.
- `-qfunctrace -qfunctrace+x` traces all functions.
- `-qfunctrace+y -qnofunctrace` traces `y` only.
- `-qfunctrace+std::vector` traces all instantiations of `std::vector`.

Related information

- For details about **#pragma nofunctrace**, see "#pragma nofunctrace" on page 361.
- For detailed information about how to implement function tracing routines in your code, as well as detailed examples and a list of rules for using them, see **Tracing functions in your code** in the *XL C Optimization and Programming Guide*.

-g

Category

Error checking and debugging

Pragma equivalent

None.

Purpose

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

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

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

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

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

Syntax

```
          ┌─0─┐
►►── -g───┤   ├──────────────────────────────────────────────────►◄
          ├─1─┤
          ├─2─┤
          ├─3─┤
          ├─4─┤
          ├─5─┤
          ├─6─┤
          ├─7─┤
          ├─8─┤
          └─9─┘
```

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.

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

- "-qdbxextra" on page 130
- "-qsymtab" on page 301
- "#pragma ibm snapshot" on page 355
- "-qlinedebug" on page 216
- "-qfullpath" on page 156
- "-O, -qoptimize" on page 236
- "-qkeepparm" on page 202

-G

Category

Output control

Pragma equivalent

None.

Purpose

Generates a shared object enabled for runtime linking.

Syntax

```
►►— -G————————————————————————————————————————————————————————►◄
```

Usage

The compiler automatically exports all global symbols from the shared object unless you specify which symbols to export by using with the **-bE:**, **-bexport:**, or **-bnoexpall** option. You can also prevent weak symbols from being exported by using the **-qnoweakexp** option. ▶ IBM Symbols that have the hidden or internal visibility attribute are not exported. IBM ◀

To save the export list to a file, use the **-qexpfile** option.

Predefined macros

None.

Related information

- “-b” on page 109
- “-brtl” on page 113
- “-qexpfile” on page 141
- “-qmkshrobj” on page 233
- “-qweakexp” on page 328
- “-qvisibility” on page 323
- “#pragma GCC visibility push, #pragma GCC visibility pop” on page 349
- Summary of compiler options by functional category: Linking
- "Shared Objects and Runtime Linking" in *AIX General Programming Concepts: Writing and Debugging Programs*
- **ld** in *AIX Commands Reference, Volume 3: i through m*

-qgenproto

Category

Portability and migration

Pragma equivalent

None.

Purpose

Produces prototype declarations from K&R function definitions or function definitions with empty parentheses, and displays them to standard output.

The compiler accepts and compiles K&R function definitions or definitions with a function declarator with empty parentheses; however, these function definitions are considered by the C standard to be obsolete (the compiler will diagnose them if you enable the **-qinfo=obs** option). When **-qgenproto** is in effect, the compiler generates the corresponding prototype declarations and displays them to standard output. You can use this option to help you identify obsolete function definitions and automatically obtain equivalent prototypes.

Syntax

```
          ┌─nogenproto──────────────────┐
►►── -q───┴─genproto───┬─────────────┬──┴──────────────────────────────────►◄
                       └─=──parmnames─┘
```

Defaults

-qnogenproto

Parameters

parmnames
Parameter names are included in the prototype. If you do not specify this suboption, parameter names will not be included in the prototype.

Predefined macros

None.

Examples

Compiling with **- qgenproto** for the following function definitions:

```
int foo(a, b)   // K&R function
  int a, b;
{
}

int faa(int i) { } // prototyped function

main() {  // missing void parameter
}
```

produces the following output on the display:

```
int foo(int, int);
int main(void);
```

Specifying **-qgenproto=parmnames** produces:

```
int foo(int a, int b);
int main(void);
```

-qhalt

Category

Error checking and debugging

Pragma equivalent

#pragma options halt

Purpose

Stops compilation before producing any object, executable, or assembler source files if the maximum severity of compile-time messages equals or exceeds the severity you specify.

Syntax

-qhalt syntax (for C)

```
                  ┌─s─┐
►►── -qhalt──=────┼─i─┼──────────────────────────────────────────────────►◄
                  ├─w─┤
                  └─e─┘
```

Defaults

-qhalt=s

Parameters

i Specifies that compilation is to stop for all types of errors: warning, error and informational. Informational diagnostics (I) are of the lowest severity.

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

e Specifies that compilation is to stop for errors (E), severe errors (S), and unrecoverable errors (U).

s Specifies that compilation is to stop for severe errors (S) and unrecoverable errors (U).

Usage

When the compiler stops as a result of the **halt** option, the compiler return code is nonzero. For a list of return codes, see "Compiler return codes" on page 18.

When **-qhalt** is specified more than once, the lowest severity level is used.

Diagnostic messages may be controlled by the **-qflag** option.

You can also instruct the compiler to stop compilation based on the number of errors of a type of severity by using the **-qmaxerr** option, which overrides **-qhalt**.

You can also use the **-qhaltonmsg** option to stop compilation according to error message number.

Predefined macros

None.

Examples

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

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

Related information

- "-qhaltonmsg"
- "-qflag" on page 145
- "-qmaxerr" on page 228

-qhaltonmsg

Category

Error checking and debugging

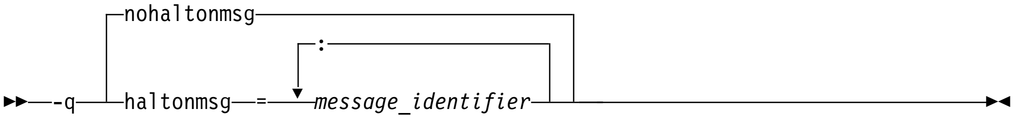
Pragma equivalent

None.

Purpose

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

Syntax



Defaults

-qnohaltonmsg

Parameters

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

`15`*`dd`*`-`*`number`*

where:

15 Is the compiler product identifier.

dd Is the two-digit code representing the compiler component that produces the message. See "Compiler message format" on page 17 for descriptions of these codes.

number
Is the message number.

Usage

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

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

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

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

Predefined macros

None.

Related information

- "-qflag" on page 145
- "-qhalt" on page 165
- "Compiler messages" on page 16
- "-qsuppress" on page 299

-qheapdebug

Category

Error checking and debugging

Pragma equivalent

None.

Purpose

Enables debug versions of memory management functions.

The compiler ships a set of "debug" versions of the standard memory management functions defined in stdlib.h (such as `_debug_calloc` and `_debug_malloc`); the header files for these functions are found in the product include directory (`/opt/IBM/xlc/13.1.3/include`). By default, the compiler uses the regular memory management functions (such as `calloc` and `malloc`) and does not preinitialize their local storage. When **-qheapdebug** is in effect, the compiler searches for header files first in the product include directory, where the debug versions of memory management functions are stored, and then in the system include directory.

Syntax

```
              ┌─noheapdebug─┐
►►── -q───────┴─heapdebug───┴──────────────────────────────────────────────►◄
```

Defaults

-qnoheapdebug

Usage

For complete information about the debug memory management functions, see "Memory debug library functions" in the *XL C Optimization and Programming Guide*.

Note: The compiler supports the memory allocation debug functions, but IBM has no plans to change or enhance these functions, and these functions will be removed in a future release. If you use these functions to debug memory problems in your programs, you can migrate to the AIX debug malloc tool to achieve equivalent functionality.

Predefined macros

__DEBUG_ALLOC__ is defined to 1 when **-qheapdebug** is in effect; otherwise, it is undefined.

Examples

To compile `myprogram.c` with the debug versions of memory management functions, enter the following command:

```
xlc -qheapdebug myprogram.c -o testing
```

Related information

- "Debugging memory heaps" in the *XL C Optimization and Programming Guide*

-qhelp

Category

Listings, messages, and compiler information

Pragma equivalent

None.

Purpose

Displays the man page of the compiler.

Syntax

```
►►— -q—help————————————————————————————————————————————————►◄
```

Usage

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

Predefined macros

None.

Related information

- “-qversion” on page 321

-qhot

Category

Optimization and tuning

Pragma equivalent

#pragma novector

#pragma nosimd

Purpose

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

The **-qhot** compiler option is a powerful alternative to hand tuning that provides opportunities to optimize loops and array language. This compiler option will always attempt to optimize loops, regardless of the suboptions you specify.

You can use the pragma directives to disable these transformations for selected sections of code.

Syntax

```
          ┌─nohot──────────────────────────────────────┐
►►── -q───┴─hot───┬──────────────────────────────────┬─┴──────────────────►◄
                  │       ┌─ :─────────────────────┐ │
                  │       ▼                        │ │
                  └─=─────┬─noarraypad─────────────┬─┘
                          ├─arraypad──┬──────────┬─┤
                          │           └─=─number─┘ │
                          │            ┌─1─┐       │
                          ├─level──=───┼─0─┼───────┤
                          │            └─2─┘       │
                          ├─vector─────────────────┤
                          ├─novector───────────────┤
                          ├─fastmath───────────────┤
                          └─nofastmath─────────────┘
```

Pragma syntax

```
►►──#──pragma──┬─novector─┬───────────────────────────────────────────────►◄
               └─nosimd───┘
```

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 (option only) | noarraypad (option only)

Permits the compiler to increase the dimensions of arrays where doing so might improve the efficiency of array-processing loops. (Because of the implementation of the cache architecture, array dimensions that are powers of two can lead to decreased cache utilization.) Specifying **-qhot=arraypad** when your source includes large arrays with dimensions that are powers of 2 can reduce cache misses and page faults that slow your array processing programs. This can be particularly effective when the first dimension is a power of 2. If you use this suboption with no *number*, the compiler will pad any arrays where it infers there may be a benefit and will pad by whatever amount it chooses. Not all arrays will necessarily be padded, and different arrays may be padded by different amounts. If you specify a *number*, the compiler will pad every array in the code.

Note: Using **arraypad** can be unsafe, as it does not perform any checking for reshaping or equivalences that may cause the code to break if padding takes place.

number **(option only)**

A positive integer value representing the number of elements by which each array will be padded in the source. The pad amount must be a positive integer value. 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 (option only)`
Performs a subset of the high-order transformations and sets the default to **novector:noarraypad:fastmath**.

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

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

`vector (option only) | novector`
When specified with **-qnostrict** and **-qignerrno**, or an optimization level of **-O3** or higher, **vector** causes the compiler to convert certain operations that are performed in a loop on successive elements of an array (for example, square root, reciprocal square root) into a call to a routine in the Mathematical Acceleration Subsystem (MASS) library in libxlopt.

The **vector** suboption supports single-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 (option only) | nofastmath (option only)`
You can use this suboption to tune your application to either use fast scalar versions of math functions or use the default versions.

You must use this suboption together with **-qignerrno**, unless **-qignerrno** is already enabled by other options.

-qhot=fastmath enables the replacement of math routines with available math routines from the XLOPT library only if **-qstrict=nolibrary** is enabled.

-qhot=nofastmath disables the replacement of math routines by the XLOPT library. **-qhot=fastmath** is enabled by default if **-qhot** is specified regardless of the hot level.

Usage

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

If you want to override the default **level** setting of **1** when using **-qsmp**, **-O4** or **-O5**, be sure to specify **-qhot=level=0** or **-qhot=level=2** *after* the other options.

The pragma directives apply only to `while`, `do while`, and `for` loops that immediately follow the placement of the directives. They have no effect on other loops that may be nested within the specified loop.

You can use the **-qreport** option in conjunction with **-qhot** or any optimization option that implies **-qhot** to produce a pseudo-C report showing how the loops were transformed. The loop transformations are included in the listing report if either the **-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 263.

Predefined macros

None.

Related information

- "-qarch" on page 102
- "-qsimd" on page 278
- "-qprefetch" on page 256
- "-qreport" on page 263
- "-qlistfmt" on page 218
- "-O, -qoptimize" on page 236
- "-qstrict" on page 294
- "-qsmp" on page 281
- *Using the Mathematical Acceleration Subsystem (MASS)* in the *XL C Optimization and Programming Guide*

-I

Category

Input control

Pragma equivalent

None.

Purpose

Adds a directory to the search path for include files.

Syntax

```
►►— -I—directory_path————————————————————————————————————————►◄
```

Defaults

See "Directory search sequence for included files" on page 12 for a description of the default search paths.

Parameters

directory_path
The path for the directory where the compiler should search for the header files.

Usage

If **-qnostdinc** is in effect, the compiler searches *only* the paths specified by the **-I** option for header files, and not the standard search paths as well. If **-qidirfirst** is in effect, the directories specified by the **-I** option are searched before any other directories.

If the **-I** directory option is specified both in the configuration file and on the command line, the paths specified in the configuration file are searched first. The **-I** directory option can be specified more than once on the command line. If you specify more than one **-I** option, directories are searched in the order that they appear on the command line.

The **-I** option has no effect on files that are included using an absolute path name.

Predefined macros

None.

Examples

To compile `myprogram.c` and search `/usr/tmp` and then `/oldstuff/history` for included files, enter:

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

Related information

- “-qstdinc” on page 292
- “-qinclude” on page 176
- “Directory search sequence for included files” on page 12
- “Specifying compiler options in a configuration file” on page 7

-qidirfirst

Category

Input control

Pragma equivalent

#pragma options [no]idirfirst

Purpose

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

Syntax

```
           ┌─noidirfirst─┐
►►── -q────┴─idirfirst───┴───────────────────────────────────────────────►◄
```

Defaults

-qnoidirfirst

Usage

This option only affects files that are included by the `#include "`*`file_name`*`"` directive or the **-qinclude** option. This option has no effect on the search order for XL 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.

The last valid pragma directive remains in effect until replaced by a subsequent pragma.

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 172
- "-qinclude" on page 176
- "-qstdinc" on page 292
- "-qc_stdinc" on page 125
- "Directory search sequence for included files" on page 12

-qignerrno

Category

Optimization and tuning

Pragma equivalent

#pragma options [no]ignerrno

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

```
         ┌─noignerrno─┐
►►── -q──┴─ignerrno───┴──────────────────────────────────────────►◄
```

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

None.

Related information

- "-O, -qoptimize" on page 236

-qignprag

Category

Language element control

Pragma equivalent

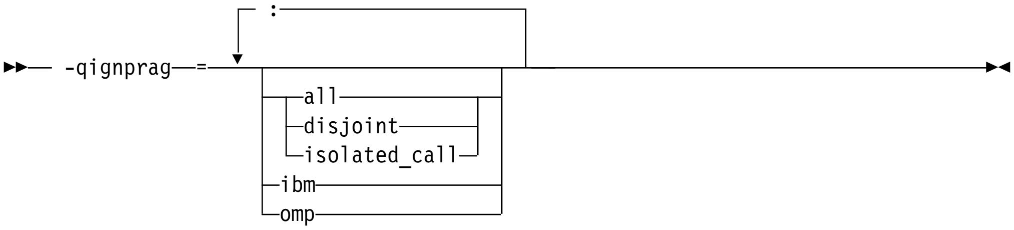
#pragma options ignprag

Purpose

Instructs the compiler to ignore certain pragma statements.

This option is useful for detecting aliasing pragma errors. Incorrect aliasing gives runtime errors that are hard to diagnose. When a runtime error occurs, but the error disappears when you use **ignprag** with the **-O** option, the information specified in the aliasing pragmas is likely incorrect.

Syntax



Defaults

Not applicable.

Parameters

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

`disjoint`
: Ignores all **#pragma disjoint** directives in the source file.

`ibm`
: Ignores all **#pragma ibm snapshot** directives and all IBM SMP directives (such as **#pragma ibm schedule**) in the source file.

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

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

Predefined macros

None.

Examples

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

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

Related information

- “#pragma disjoint” on page 345
- “-qisolated_call” on page 199
- “#pragma ibm snapshot” on page 355
- “Pragma directives for parallel processing” on page 380

-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

```
          ┌─noinclude──────────┐
►►──-q────┴─ include──=──file──┴──────────────────────────────────────────────►◄
```

Defaults

-qnoinclude

Parameters

file

The absolute or relative path and name of the header file to be included in the compilation units being compiled. If *file* is specified with a relative path, the search for it follows the sequence described in "Directory search sequence for included files" on page 12.

Usage

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

The **-qinclude** option applies only to the files specified in the same compilation in which the option is specified. It is not passed to any compilations that occur during the link step, nor to any implicit compilations.

When the option is specified multiple times in an invocation, the header files are included in order of appearance on the command line. If the same header file is specified multiple times with this option, the header is treated as if included multiple times by `#include` directives in the source file, in order of appearance on the command line.

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

Any pragma directives that must appear before noncommentary statements in a source file will be affected; you cannot use **-qinclude** to include files if you need to preserve the placement of these pragmas.

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

- If you generate a listing file with **-qsource**, the header files included by **-qinclude** do not appear in the source section of the listing. Use **-qshowinc=usr** or **-qshowinc=all** in conjunction with **-qsource** if you want these header files to appear in the listing.
- After searching the directory from which the compiler was invoked, **-qinclude** searches additional search paths added to the search chain by the **-I** option. You can specify the **-I** option before or after the **-qinclude** option.
- Files specified with **-qinclude** are included as dependencies in the **-qmakedep** output. However, the paths are different in the dependency file for the **-qinclude** option and the `#include` directive, because the files specified with the **-qinclude** option are searched in the invocation path first, whereas files included by the `#include` directive are not.

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

Predefined macros

None.

Examples

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

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

Related information

- "Directory search sequence for included files" on page 12

-qinfo

Category

Error checking and debugging

Pragma equivalent

#pragma options [no]info, #pragma info

Purpose

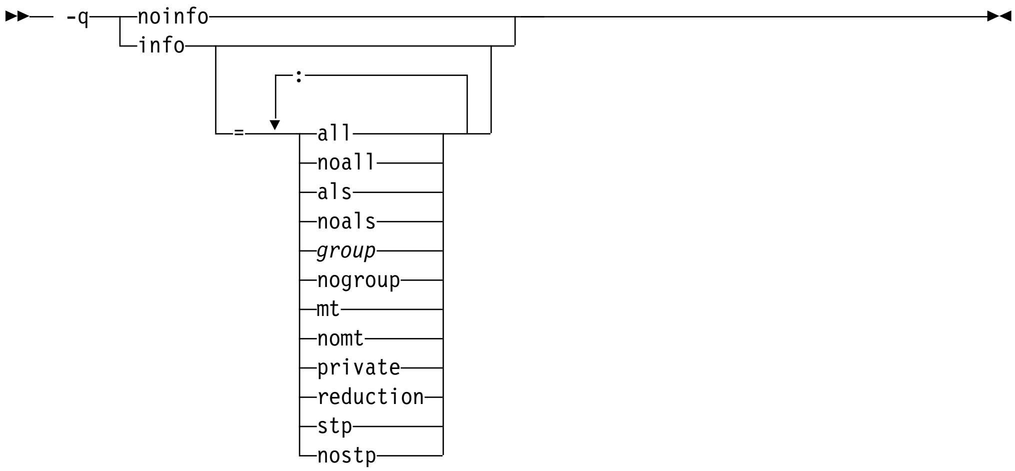
Produces or suppresses groups of informational messages.

The messages are written to standard output and, optionally, to the listing file if one is generated. The compiler does not issue messages for the following files:

- Files in the standard search paths for compiler and system header files. The standard search paths are affected by the following compiler options:
  - "-qstdinc" on page 292
  - "-qc_stdinc" on page 125
- Files that are ultimately included by the files in the standard search paths for compiler and system header files.

Syntax

Option syntax



Pragma syntax

```
                        ┌─,──────────┐
                        ▼            │
►►─#─pragma─info─(────┬─all──────┬───┴─)──────────────────────────►◄
                      ├─none─────┤
                      ├─als──────┤
                      ├─noals────┤
                      ├─group────┤
                      ├─nogroup──┤
                      ├─mt───────┤
                      ├─nomt─────┤
                      ├─private──┤
                      ├─reduction┤
                      └─restore──┘
```

Defaults

-qnoinfo

Parameters

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

noall (option only)
Disables all diagnostic messages for all groups.

none (pragma only)
Disables all diagnostic messages for all groups.

als
Enables reporting possible violations of the ANSI aliasing rule in effect.

noals
Disables reporting possible aliasing-rule violations.

group | **no***group*
Enables or disables specific groups of messages, where *group* can be one or more of:

group
Type of informational messages returned or suppressed.

cmp | nocmp
Possible redundancies in `unsigned` comparisons.

cnd | nocnd
Possible redundancies or problems in conditional expressions.

cns | nocns
Operations involving constants.

cnv | nocnv
Conversions.

dcl | nodcl
Consistency of declarations.

eff | noeff
Statements and pragmas with no effect.

enu | noenu
Consistency of enum variables.

ext | noext
Unused external definitions.

gen | nogen
General diagnostic messages.

gnr | nognr
Generation of temporary variables.

got | nogot
Use of goto statements.

ini | noini
Reports array initializers that partially initialize their arrays. If an array is partially initialized, elements that are not initialized receive the value 0 of the appropriate type.

lan | nolan
Language level effects.

obs | noobs
Obsolete features.

ord | noord
Unspecified order of evaluation.

par | nopar
Unused parameters.

por | nopor
Nonportable language constructs.

ppc | noppc
Possible problems with using the preprocessor.

ppt | noppt
Trace of preprocessor actions.

pro | nopro
Missing function prototypes.

rea | norea
Code that cannot be reached.

ret | noret
Consistency of return statements.

trd | notrd
Possible truncation or loss of data or precision.

tru | notru
Variable names truncated by the compiler.

trx | notrx
Hexadecimal-floating point constants rounding.

uni | nouni
Uninitialized variables. The **-qinfo=uni** option enforces the coding style that a variable must be initialized in its declaration.

upg | noupg
Generates messages describing new behaviors of the current compiler release as compared to the previous release.

use | nouse
Unused auto and static variables.

zea | nozea
Zero-extent arrays.

mt | nomt

Reports potential synchronization issues in parallel code. This suboption detects the Global Thread Flag pattern where one thread uses a shared volatile flag variable to notify other threads that it has completed a computation and stored its result to memory. Other threads check the flag variable in a loop that does not change the flag variable. When the value of the flag variable changes, the waiting threads access the computation result from memory. The PowerPC storage model requires synchronization before the flag variable is set in the first thread, and after the flag variable is checked in the waiting threads. Synchronization can be done by a synchronization built-in function.

The type of synchronization directives you need to use depends on your code. Usually, it is enough to use the `__lwsync` function, as it preserves the storage access order to system memory. However, if the loops in the waiting threads are written in such a way that might cause instruction prefetching to start executing code that accesses the computation result before the flag variable is updated, a call to a function like `__isync` is needed to preserve order. Such patterns are typically as follows:

```
gotosleep: sleep(value);
   if (!flag) goto gotosleep;
   // A call to the __isync function is needed here.
   x = shared_computation_result;
```

Some patterns that do not require synchronization are similar to the patterns described above. The messages generated by this suboption are only suggestions about potential synchronization issues.

To use the **-qinfo=mt** suboption, you must enable the **-qthreaded** option and specify at least one of the following options:

- **-O3**
- **-O4**
- **-O5**
- **-qipa**
- **-qhot**
- **-qsmp**

The default option is **-qinfo=nomt**.

private
This suboption is deprecated. **-qreport** replaces it. For details, see "-qreport" on page 263 and the "Deprecated options" on page 91 section in the *XL C Compiler Reference*.

reduction
This suboption is deprecated. **-qreport** replaces it. For details, see "-qreport" on page 263 and the "Deprecated options" on page 91 section in the *XL C Compiler Reference*.

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

restore (pragma only)

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

Usage

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

Specifying **-qnoinfo** is equivalent to **-qinfo=noall**.

Consider the following when enabling the reporting of aliasing-rule violations:

- **-qalias=ansi** must be set before reporting of aliasing-rule violations (**-qinfo=als**) can occur.
- Any level of optimization or inlining implies **-qinfo=noals** and a warning will be issued when **-qinfo=als** is explicitly specified.
- Diagnostics are heuristic and may emit false positives. Points-to analysis cannot be evaluated deterministically in static compilation. The points-to analysis used for diagnostics is evaluated in a context-and-flow, insensitive manner. The sequence of traceback messages in diagnostics is such that if executed in the order specified, the indirect expression will point to the offending object. If that execution sequence cannot occur in the application, the diagnostic is a false positive. (See the **Examples** section for the types of diagnostics that can occur.)

Predefined macros

None.

Examples

To compile `myprogram.c` to produce informational messages about all items except conversions and unreached statements, enter the following command:

```
xlc myprogram.c -qinfo=all -qinfo=nocnv:norea
```

The following example shows code constructs that the compiler detects when the code is compiled with **-qinfo=cnd:eff:got:obs:par:pro:rea:ret:uni** in effect:

```
#define COND 0

void faa() // Obsolete prototype (-qinfo=obs)
{
   printf("In faa\n"); // Unprototyped function call (-qinfo=pro)
}

int foo(int i, int k)
{
   int j; // Uninitialized variable (-qinfo=uni)

   switch(i) {
   case 0:
   i++;
   if (COND) // Condition is always false (-qinfo=cnd)
      i--;   // Unreachable statement (-qinfo=rea)
   break;

   case 1:
      break;
      i++;   // Unreachable statement (-qinfo=rea)
   default:
      k = (i) ? (j) ? j : i : 0;
}

   goto L;   // Use of goto statement (-qinfo=got)
   return 3; // Unreachable statement (-qinfo=rea)
L:
   faa(); // faa() does not have a prototype (-qinfo=pro)

// End of the function may be reached without returning a value
// because of there may be a jump to label L (-qinfo=ret)

} //Parameter k is never referenced (-qinfo=ref)

int main(void) {
({ int i = 0; i = i + 1; i; }); // Statement does not have side effects (-qinfo=eff)

 return foo(1,2);
}
```

In the following example, the **#pragma info(eff, nouni)** directive preceding `MyFunction1` instructs the compiler to generate messages identifying statements or pragmas with no effect, and to suppress messages identifying uninitialized variables. The **#pragma info(restore)** directive preceding `MyFunction2` instructs the compiler to restore the message options that were in effect before the **#pragma info(eff, nouni)** directive was specified.

```
#pragma info(eff, nouni)
int MyFunction1()
{
  .
  .
  .

}

#pragma info(restore)
int MyFunction2()
{
  .
  .
  .

}
```

The following example shows a valid diagnostic for an aliasing violation:

```
t1.c:
int main() {
  short s = 42;
  int *pi = (int*) &s;
  *pi = 63;
  return 0;
}
xlC -qinfo=als t1.c
"t1.c", line 4.3: 1540-0590 (I) Dereference may not conform to the current
                                aliasing rules.
"t1.c", line 4.3: 1540-0591 (I) The dereferenced expression has type "int".
                                "pi" may point to "s" which has incompatible
                                type "short".
"t1.c", line 4.3: 1540-0592 (I) Check assignment at line 3 column 11 of t1.c.
```

In the following example, the analysis is context insensitive in that the two calls to floatToInt are not distinguished. There is no aliasing violation in this example, but a diagnostic is still issued.

```
t2.c:
int* floatToInt(float *pf) { return (int*)pf; }

int main() {
  int i;
  float f;
   int* pi = floatToInt((float*)*&i));
   floatToInt(&f;)
   return *pi;
}
```

```
xlC -qinfo=als t2.c
"t2.c", line 8.10: 1540-0590 (I) Dereference may not conform to the current
                                  aliasing rules.
"t2.c", line 8.10: 1540-0591 (I) The dereferenced expression has type "int".
                                  "pi" may point to "f"
                                  which has incompatible type "float".
"t2.c", line 8.10: 1540-0592 (I) Check assignment at line 7 column 14 of t2.c.
"t2.c", line 8.10: 1540-0592 (I) Check assignment at line 1 column 37 of t2.c.
"t2.c", line 8.10: 1540-0592 (I) Check assignment at line 6 column 11 of t2.c.
```

```
t3.c:
int main() {
  float f;
  int i = 42;
  int *p = (int*) &f;
  p = &i;
  return *p;
}

xlC -qinfo=als t3.c
"t3.c", line 6.10: 1540-0590 (I) Dereference may not conform to
        the current aliasing rules.
"t3.c", line 6.10: 1540-0591 (I) The dereferenced expression has
        type "int". "p" may point to "f", which has incompatible
        type "float".
"t3.c", line 6.10: 1540-0592 (I) Check assignment at line 4 column
        10 of t3.c.
```

To compile sync.c to produce informational messages about potential synchronization issues in parallel code, enter the following command:

```
 xlc_r -O3 -qinfo=mt sync.c
```

Suppose that sync.c contains the following code:

```
#include <unistd.h>
#include <stdio.h>
#include <pthread.h>

volatile int done;          /* shared flag */
volatile int result;        /* shared result */

void *setter(void *id)
{
  sleep(5);
  result = 7;
  /* Need __lwsync(); */
  done = 1;                 /* line 13 */
}

void *waiter(void *id)
{
  while (!done)             /* line 18 */
  {
    sleep(1);
  }
  /* need __lwsync(); */
  printf("%d\n", result);
}

int main()
{
  pthread_t threads[2];
  pthread_attr_t attr;
  int result;

  result = pthread_create(&threads[0], NULL, waiter, NULL);
  if (result != 0) exit(2);
  result = pthread_create(&threads[1], NULL, setter, NULL);
  if (result != 0) exit(3);
  pthread_join(threads[0], NULL);
  pthread_join(threads[1], NULL);

  return 0;
}
```

The compiler issues the following informational messages:

```
1586-669 (I) "sync.c", line 18: If this loop is used as a synchronization
point, additional synchronization via a directive or built-in function might
be needed.

1586-670 (I) "sync.c", line 13: If this statement is used as a synchronization
point, additional synchronization via a directive or built-in function might
be needed.
```

The following function `ini.c` partially initialized array `a[3]`. To compile `ini.c` to produce an informational message about this issue, enter the following command:

```
xlc -qinfo=ini ini.c -c
```

Suppose that `ini.c` contains the following code:

```
int a[3] = {1};
```

The compiler issues the following informational message:

```
"ini.c", line 1.10: 1506-446 (I) Array element(s) [1] ...
[2] will be initialized with a default value of 0.
```

The following function `factorial.c` does not initialize `result` when `n`<1. With **-qinfo=unset** at **-qnoopt**, this issue is not detected. To compile `factorial.c` to produce informational messages about the uninitialized variable `result`, enter the following command:

```
xlc -qinfo=unset -O 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:

```
1500-099: (I) "factorial.c", line 8: "result" might be used before it is set.
```

Related information

- “-qflag” on page 145
- “-qreport” on page 263
- “-qstackprotect” on page 291
- “Synchronization functions” on page 462
- For a list of deprecated options, see the “Deprecated options” on page 91 section in the *XL C Compiler Reference*.

- For more information about synchronization and the PowerPC storage model, see the article at http://www.ibm.com/developerworks/systems/articles/powerpc.html.

-qinitauto

Category

Error checking and debugging

Pragma equivalent

#pragma options [no]initauto

Purpose

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

Syntax

```
           ┌─noinitauto──────────────┐
►►── -q────┴─initauto──=──hex_value──┴──────────────────────────────────────►◄
```

Defaults

-qnoinitauto

Parameters

hex_value
A one- to eight-digit hexadecimal number.

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

Usage

The -qinitauto option provides the following benefits:

- Setting *hex_value* to zero ensures that all automatic variables that are not explicitly initialized when declared are cleared before they are used.
- You can use this option to initialize variables of real or complex type to a signaling or quiet NaN, which helps locate uninitialized variables in your program.

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

Restrictions:

- Objects that are equivalenced, structure components, and array elements are not initialized individually. Instead, the entire storage sequence is initialized collectively.
- The **-qinitauto=hex_value** option does not initialize variable length arrays or memory allocated through the `__alloca` function.

Predefined macros

- __INITAUTO__ is defined to the least significant byte of the *hex_value* that is specified on the **-qinitauto** option or pragma; otherwise, it is undefined.
- __INITAUTO_W__ is defined to the byte *hex_value*, repeated four times, or to the word *hex_value*, which is specified on the **-qinitauto** option or pragma; otherwise, it is undefined.

For example:

- For option -qinitauto=0xABCD, the value of __INITAUTO__ is 0xCDu, and the value of __INITAUTO_W__ is 0x0000ABCDu.
- For option -qinitauto=0xCD, the value of __INITAUTO__ is 0xCDu, and the value of __INITAUTO_W__ is 0xCDCDCDCDu.

Examples

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

```
#include <stdio.h>

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

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

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

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

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

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

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

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

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

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

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

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

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

-qinlglue

Category

Object code control

Pragma equivalent

#pragma options [no]inlglue

Purpose

When used with **-O2** or higher optimization, inlines glue code that optimizes external function calls in your application.

Glue code or , 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

```
         ┌─noinlglue─┐
►►── -q──┴─inlglue───┴──────────────────────────────────────────────►◄
```

Defaults

- **-qnoinlglue** when **-q32** is in effect
- **-qinlglue** when **-q64** is in effect
- **-qinlglue** when **-qtune=pwr4** and above, **-qtune=ppc970**, **-qtune=auto**, or **-qtune=balanced** is in effect.

Usage

If you use the **-qtune** option with any of the suboptions that imply **-qinlglue** and you want to disable inlining of glue code, make sure to specify **-qnoinlglue** as well.

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 122
- "-qprocimported, -qproclocal, -qprocunknown" on page 260
- "-qtune" on page 310

-qinline

Category

Optimization and tuning

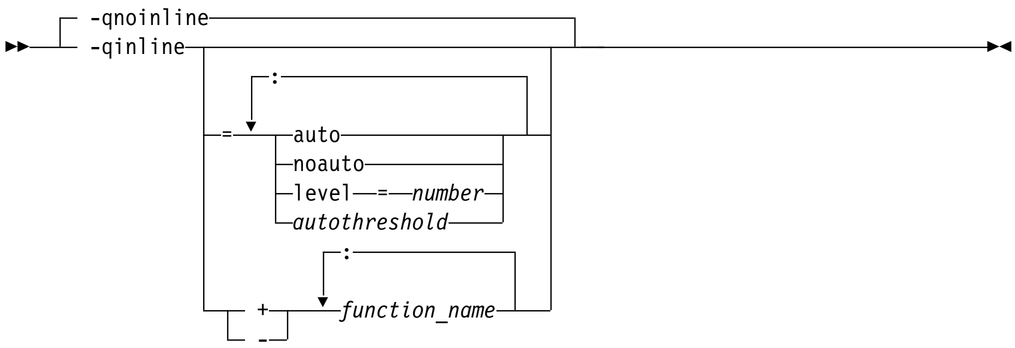
Pragma equivalent

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 **-qnoinline** at the **-O0** or **-qnoopt** optimization level, or **-qinline=noauto:level=5** at the **-O2** 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.

autothreshold
Represents the largest number of executable statements that a function can include when the function is to be inlined. The value for *autothreshold* must be a positive integer. The default value for *autothreshold* is 20. If you specify a value of 0, only functions that are specified with IBM the `always_inline` or `__always_inline__` attribute IBM or specified after **-qinline+** are inlined. In the following example, three executable statements are included in the `increment` function.

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

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.

Usage

You can specify **-qinline** with any optimization level of **-O2**, **-O3**, **-O4**, or **-O5** to enable inlining of functions, including those functions that are declared with the `inline` specifier.

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.

Note:

- **-qinline** replaces **-Q** and its suboptions.
- **-Q**, **-Q!**, **-Q**=*threshold*, **-Q+***name*, and **-Q-***name* are all deprecated options and suboptions.
- **-qipa=inline** and all of its associated suboptions are deprecated. **-qinline** replaces them all.

Predefined macros

None.

Examples

Example 1

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

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

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

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

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

You can specify an inlining level 6 - 10 to achieve more aggressive automatic inlining. For example:

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

If automatic inlining is already enabled by default and you want to specify an inlining level of 7, you enter:

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

Example 2

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

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

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

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

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

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

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

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

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

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

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

Related information

- "-g" on page 160
- "-qipa" on page 193

- “-O, -qoptimize” on page 236
- "The inline function specifier" in the *XL C Language Reference*
- "always_inline (IBM extension)" in the *XL C Language Reference*
- For a list of deprecated compiler options, see Deprecated options

-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.

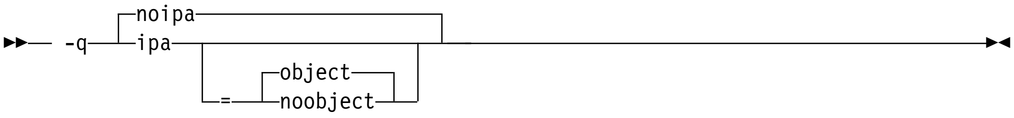
You can generate relinkable objects while preserving IPA information by specifying **-r -qipa=relink**. This creates a nonexecutable package that contains all object files. By using this suboption, you can postpone linking until the very last stage.

If you want to use your own archive files, you can use the **ar** tool and set the *XL_AR* environment variable to point to its location. If you do not specify a location, the compiler sets the environment variable according to the information contained in the configuration file.

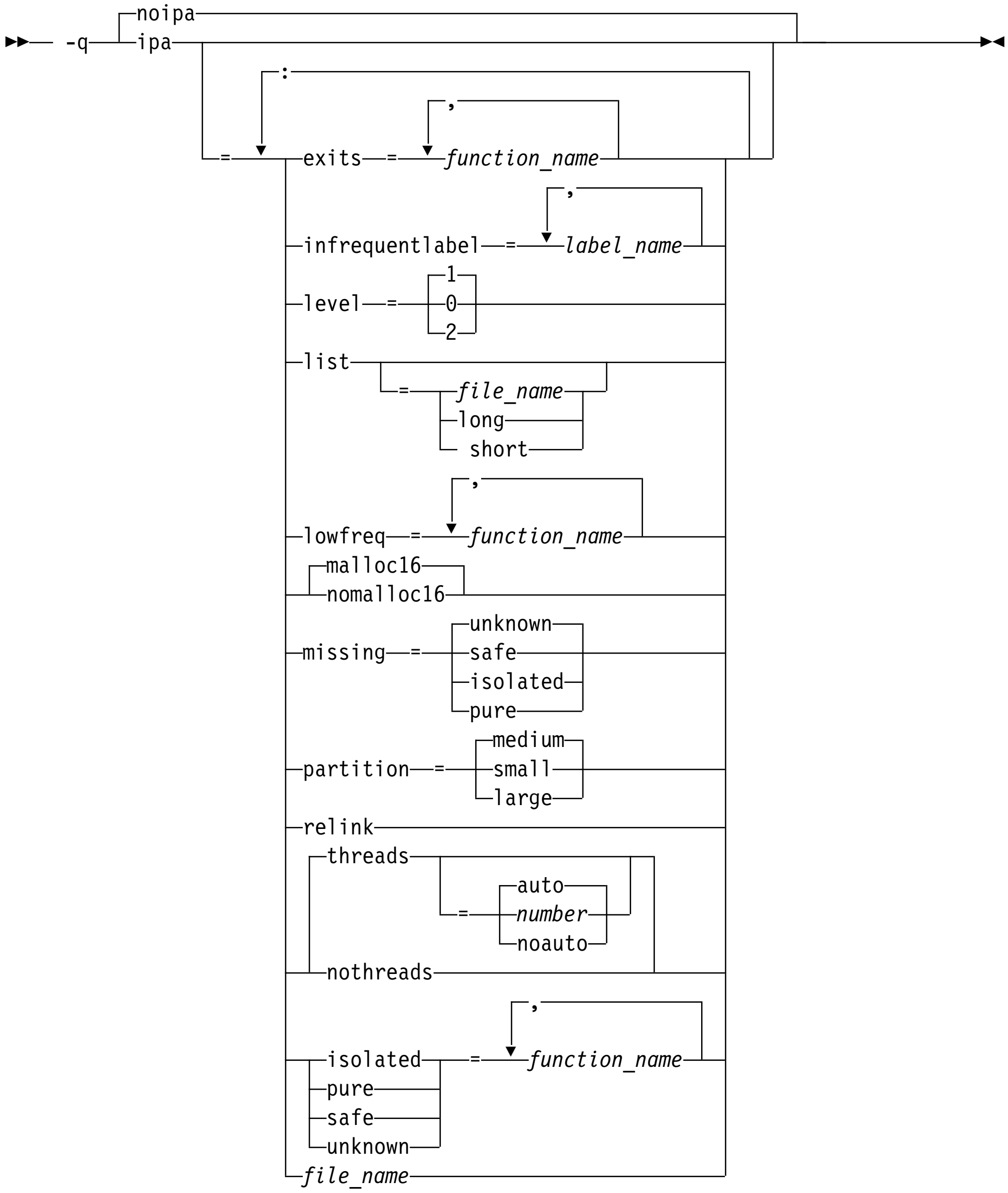
Note:
- This suboption does not link the objects; instead, it only aggregates them. As a result, the compiler does not report any error or warning messages; furthermore, the compiler ignores linker or binder options when you use this suboption.
- You must use the **-r** suboption with **-qipa=relink**. Without **-r**, **-qipa=relink** is ignored.

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.

malloc16 | nomalloc16

Informs the compiler that the dynamic memory allocation routines will return 16-byte aligned memory addresses. The compiler can then optimize the code based on that assertion.

In 64-bit mode, AIX always returns 16-byte aligned addresses and therefore by default **-qipa=malloc16** is in effect. You can use **-qipa=nomalloc16** to override the default setting.

Note: You must make sure that the executables generated with **-qipa=malloc16** run in an environment in which dynamic memory allocations return 16-byte aligned addresses, otherwise, wrong results can be generated. For example, in 32-bit mode, addresses are not 16-byte aligned. In this case, you must set the MALLOCALIGN=16 runtime environment variable.

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.

relink

Creates relinkable objects by packaging them into a nonexecutable file. When using this suboption, you must also use the **-r** option along with it. Otherwise, the compiler ignores **-qipa=relink**.

Note:

- If you use**-qipa=noobject** (either directly or indirectly) and use the **relink** suboption, you must link the resulting object files with **-qipa**. Otherwise, unresolved references to your object files can occur.
- You might indirectly use **-qipa=noobject** if you link and compile your object files in one step. In addition, you cannot use the shared objects with **-qipa=relink**, they must be used at the last link step together with the prelink output.

safe

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

threads | nothreads

Runs portions of the IPA optimization process during pass 2 in parallel threads, which can speed up the compilation process on multi-processor systems. Valid suboptions for the **threads** suboption are one of the following suboptions:

auto | noauto

When **auto** is in effect, the compiler selects a number of threads heuristically based on machine load. When **noauto** is in effect, the compiler creates one thread per machine processor.

number

Instructs the compiler to use a specific number of threads. *number* can be any integer value in the range of 1 to 32 767. However, *number* is effectively limited to the number of processors available on your system.

Specifying **threads** with no suboptions implies **-qipa=threads=auto**.

unknown

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

file_name

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

The file format is shown as follows:

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

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

Note:
- **-qipa=inline** and all of its associated suboptions are deprecated. **-qinline** replaces them all. For details, see "-qinline" on page 189 and "Deprecated options" on page 91.
- As of the V9.0 release of the compiler, the **pdfname** suboption is deprecated; you should use **-qpdf1=*pdfname*** or **-qpdf2=*pdfname*** in your new applications. See "-qpdf1, -qpdf2" on page 247 for details.

Usage

Specifying **-qipa** automatically sets the optimization level to **-O2**. For additional performance benefits, you can also specify the **-qinline** option. The **-qipa** option extends the area that is examined during optimization and inlining from a single function to multiple functions (possibly in different source files) and the linkage between them.

If any object file used in linking with **-qipa** was created with the **-qipa=noobject** option, any file containing an entry point (the main program for an executable program, or an exported function for a library) must be compiled with **-qipa**.

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

You can use **-r -qipa=relink** to create a relinkable package that contains all object files without generating an executable program. If you want to use your archive files, set the path to your **ar** tool using the *XL_AR* environment variable.

Some symbols which are clearly referenced or set in the source code may be optimized away by IPA, and may be lost to **debug**, **dump**, 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 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
```

The following example demonstrates how you can create a relinkable package that includes your object files:

```
xlc -O5 -o -r -qipa=relink result obj1.o obj2.o obj3.o

ls -l result

-rw-r--r-- result

xlc -O5 -o res result obj4.o obj5.o
```

Here is how you can generate a relinkable package using your own archive files:

```
ar -X64 -r arch1.a object11.o object12.o

ar -X64 -r arch2.a object21.o object22.o

xlc -O5 -o -r -qipa=relink -q64 result obj1.o obj2.o obj3.o arch1.a arch2.a
xlc -O5 -o res result obj4.o obj5.o
```

Related information

- “-qinline” on page 189
- “-qisolated_call”
- “-qlibmpi” on page 215
- “#pragma execution_frequency” on page 346
- -qpdf1, -qpdf2
- -r
- “-S” on page 271
- Deprecated options
- "Optimizing your applications" in the *XL C Optimization and Programming Guide*
- Runtime environment variables

-qisolated_call

Category

Optimization and tuning

Pragma equivalent

#pragma options isolated_call, #pragma isolated_call

Purpose

Specifies functions in the source file that have no side effects other than those implied by their parameters.

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

- Accessing a volatile object
- Modifying an external object
- Modifying a static object
- Modifying a file
- Accessing a file that is modified by another process or thread
- Allocating a dynamic object, unless it is released before returning
- Releasing a dynamic object, unless it was allocated during the same invocation
- Changing system state, such as rounding mode or exception handling
- Calling a function that does any of the above

Marking a function as isolated indicates to the optimizer that external and static variables cannot be changed by the called function and that pessimistic references to storage can be deleted from the calling function where appropriate. Instructions can be reordered with more freedom, resulting in fewer pipeline delays and faster execution in the processor. Multiple calls to the same function with identical parameters can be combined, calls can be deleted if their results are not needed, and the order of calls can be changed.

Syntax

Option syntax

```
                          ┌─:────────┐
                          ▼          │
►►── -q──isolated_call──=────function─┴────────────────────────────────►◄
```

Pragma syntax

```
►►──#──pragma──isolated_call──(──function──)───────────────────────────►◄
```

Defaults

Not applicable.

Parameters

function
: The name of a function that does not have side effects or does not rely on functions or processes that have side effects. *function* is a primary expression that can be an identifier. An identifier must be of type function or a `typedef` of function.

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.

The **#pragma options isolated_call** directive must be placed at the top of a source file, before any statements. The **#pragma isolated_call** directive can be placed at any point in the source file, before or after calls to the function named in the pragma.

The **-qignprag** compiler option causes aliasing pragmas to be ignored; you can use **-qignprag** to debug applications containing the **#pragma isolated_call** directive.

Predefined macros

None.

Examples

To compile `myprogram.c`, specifying that the functions `myfunction(int)` and `classfunction(double)` do not have side effects, enter:

```
xlc myprogram.c -qisolated_call=myfunction:classfunction
```

The following example shows you when to use the **#pragma isolated_call** directive (on the `addmult` function). It also shows you when not to use it (on the `same` and `check` functions):

```
#include <stdio.h>
#include <math.h>

int addmult(int op1, int op2);
#pragma isolated_call(addmult)

/* This function is a good candidate to be flagged as isolated as its */
/* result is constant with constant input and it has no side effects. */
int addmult(int op1, int op2) {
  int rslt;

  rslt = op1*op2 + op2;
  return rslt;
}

/* The function 'same' should not be flagged as isolated as its state */
/* (the static variable delta) can change when it is called. */
int same(double op1, double op2) {
  static double delta = 1.0;
  double temp;

  temp = (op1-op2)/op1;
  if (fabs(temp) < delta)
    return 1;
  else {
    delta = delta / 2;
    return 0;
  }
}

/* The function 'check' should not be flagged as isolated as it has a */
/* side effect of possibly emitting output. */
int check(int op1, int op2) {
  if (op1 < op2)
    return -1;
  if (op1 > op2)
    return 1;
  printf("Operands are the same.\n");
  return 0;
}
```

Related information

- "-qignprag" on page 175

-qkeepparm

Category

Error checking and debugging

Pragma equivalent

None.

Purpose

When used with **-O2** or higher optimization, specifies whether procedure parameters are stored on the stack.

A function usually stores its incoming parameters on the stack at the entry point. However, when you compile code with optimization options enabled, the compiler may remove these parameters from the stack if it sees an optimizing advantage in doing so. When **-qkeepparm** is in effect, parameters are stored on the stack even when optimization is enabled. When **-qnokeepparm** is in effect, parameters are removed from the stack if this provides an optimization advantage.

Syntax

```
           ┌─nokeepparm─┐
►►── -q────┴─keepparm───┴──────────────────────────────────────────────►◄
```

Defaults

-qnokeepparm

Usage

Specifying **-qkeepparm** that the values of incoming parameters are available to tools, such as debuggers, by preserving those values on the stack. However, this may negatively affect application performance.

Predefined macros

None.

Related information

- "-O, -qoptimize" on page 236

-qkeyword

Category

Language element control

Pragma equivalent

None

Purpose

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

Syntax

```
           ┌─keyword───┐
►►── -q────┴─nokeyword─┴───=──keyword_name──────────────────────────────►◄
```

Defaults

By default, all the built-in keywords defined in the C language standard are reserved as keywords.

Usage

You cannot add keywords to the language with this option. However, you can use **-qnokeyword=***keyword_name* to disable built-in keywords, and use **-qkeyword=***keyword_name* to reinstate those keywords.

This option can be used with the following C keywords:

- asm
- inline
- restrict
- typeof

Note: `asm` is not reserved as a keyword at the **stdc89** or **stdc99** language level.

Predefined macros

- __C99_INLINE is defined to 1 when **-qkeyword=inline** is in effect.
- __C99_RESTRICT is defined to 1 when **-qkeyword=restrict** is in effect.
- __IBM_GCC_ASM is defined to 1 when **-qkeyword=asm** is in effect.
- __IBM__TYPEOF__ is defined to 1 when **-qkeyword=typeof** is in effect.

Examples

You can reinstate `typeof` with the following invocation:

```
xlc -qkeyword=typeof
```

-l

Category

Linking

Pragma equivalent

None.

Purpose

Searches for the specified library file. For static and dynamic linking, the linker searches for *libkey.a*. For runtime linking with the **-brtl** option, the linker searches for *libkey.so*, and then *libkey.a* if *libkey.so* is not found.

Syntax

```
►►— -l—key————————————————————————————————————————————————————►◄
```

Defaults

The compiler default is to search only some of the compiler runtime libraries. The default configuration file specifies the default library names to search for with the **-l** compiler option, and the default search path for libraries with the **-L** compiler option.

The C 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** or **-Z** option. See "-B" on page 110, "-brtl" on page 113, and "-b" on page 109 for information about specifying the types of libraries that are searched (for static or dynamic linking).

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.a` that is found in the `/usr/mylibdir` directory, enter the following command:

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

Assume that the `libmyrtlibrary.so` library has been compiled for runtime linking via the **-G** option and is located in the `/usr/mylibdir` directory. To compile `myrtprogram.c` and link it with library `libmyrtlibrary.so`, enter the following command:

```
xlc -brtl myrtprogram.c -lmyrtlibrary -L/usr/mylibdir
```

Related information

- "-L"
- "-b" on page 109
- "-brtl" on page 113
- "-Z" on page 333
- "Specifying compiler options in a configuration file" on page 7

-L

Category

Linking

Pragma equivalent

None.

Purpose

Searches the directory path for library files specified by the **-l** option.

Syntax

```
►►— -L—directory_path——————————————————————————————————————►◄
```

Defaults

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

Parameters

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

Usage

When you link shared libraries into an executable, specifying the paths to the libraries with the **-L** option during the link also embeds the path information in the executable, so the shared libraries can be correctly located at run time. If you do not specify any paths with **-L** during this link and you additionally prevent the compiler from automatically passing **-L** arguments to the linker by using the **-bnolibpath** linker option, only paths that are specified by the LIBPATH environment variable are embedded in the executable file.

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.

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 204

-qlanglvl

This topic includes the following information:

- "Category"
- "Pragma equivalent"
- "Purpose"
- "Syntax"
- "Defaults" on page 207
- " Parameters " on page 207
- "Usage" on page 209
- "Predefined macros" on page 209

Category

Language element control

Pragma equivalent

#pragma options langlvl, #pragma langlvl

Purpose

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

Syntax

Option syntax

```
                   ┌─:──────────────────┐
                   │ ┌─extc99───────────┐ │
►►──-q──langlvl──=─▼─┼─classic──────────┼─┴──────────────────────────►◄
                     ├─extc1x───────────┤
                     ├─extc89───────────┤
                     ├─extended─────────┤
                     ├─saa──────────────┤
                     ├─saal2────────────┤
                     ├─stdc89───────────┤
                     ├─stdc99───────────┤
                     └─feature_suboption┘
```

Pragma syntax

```
                            ┌─extc99───┐
►►──#──pragma──langlvl──(───┼─classic──┼──)──────────────────────────►◄
                            ├─extc1x───┤
                            ├─extc89───┤
                            ├─extended─┤
                            ├─saa──────┤
                            ├─saal2────┤
                            ├─stdc89───┤
                            └─stdc99───┘
```

Defaults

- The default is set according to the command used to invoke the compiler:
  - **-qlanglvl=extc99:ucs** for the **xlc** and related invocation commands
  - **-qlanglvl=extended:noucs** for the **cc** and related invocation commands
  - **-qlanglvl=stdc89:noucs** for the **c89** and related invocation commands
  - **-qlanglvl=stdc99:ucs** for the **c99** and related invocation commands
-

Parameters

The following are the **-qlanglvl**/**#pragma langlvl** parameters for C language programs:

classic

Allows the compilation of nonstandard programs, and conforms closely to the K&R level preprocessor. This language level is not supported by the AIX V5.1 and higher system header files, such as math.h. If you use the AIX V5.1 or higher system header files, consider compiling your program to the **stdc89** or **extended** language levels.

For details, see "Differences between the classic language level and all other standard-based language levels" on page 209.

▶ C11 **extc1x**

Compilation is based on the C11 standard, invoking all the currently supported C11 features and other implementation-specific language extensions.

For more information about these C11 features, see Extensions for C11 compatibility in the *XL C Language Reference*.

Note: IBM supports selected features of C11, known as C1X before its ratification. 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 C11 features is complete, including the support of a new C11 standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the C11 features. C11

extc89
Compilation conforms to the ANSI C89 standard, and accepts implementation-specific language extensions.

extc99
Compilation conforms to the ISO C99 standard, and accepts implementation-specific language extensions.

extended
Provides compatibility with the RT compiler and **classic**. This language level is based on C89.

saa
Compilation conforms to the current SAA C CPI language definition. This is currently SAA C Level 2.

saal2
Compilation conforms to the SAA C Level 2 CPI language definition, with some exceptions.

stdc89
Compilation conforms strictly to the ANSI C89 standard, also known as ISO C90.

stdc99
Compilation conforms strictly to the ISO C99 standard.

Note: Not all operating system releases support the header files and runtime library required by C99.

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

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

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

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

ucs | noucs (option only)
Controls whether Unicode characters are allowed in identifiers, string literals and character literals in program source code. This suboption is enabled by default when **stdc99** or **extc99** is in effect. For details on the Unicode character set, see "The Unicode standard" in the *XL C Language Reference*.

The following **-qlanglvl** suboptions are accepted but ignored by the C compiler. Use **extended | extc99 | extc89** to enable the functions that these suboptions imply. For other language levels, the functions implied by these suboptions are disabled.

[no]gnu_assert
GNU C portability option.

[no]gnu_explicitregvar
GNU C portability option.

[no]gnu_include_next
GNU C portability option.

[no]gnu_locallabel
GNU C portability option.

[no]gnu_warning
GNU C portability option.

Usage

Since the pragma directive makes your code non-portable, it is recommended that you use the option rather than the pragma. If you do use the pragma, it must appear before any noncommentary lines in the source code. Also, because the directive can dynamically alter preprocessor behavior, compiling with the preprocessing-only options may produce results different from those produced during regular compilation.

Predefined macros

See "Macros related to language levels" on page 408 for a list of macros that are predefined by **-qlanglvl** suboptions.

Related information

- "-qsuppress" on page 299

Differences between the classic language level and all other standard-based language levels

This topic outlines the differences between the **classic** language level and all other standard-based language levels.

Tokenization

Tokens introduced by macro expansion may be combined with adjacent tokens in some cases. Historically, this was an artifact of the text-based implementations of older preprocessors, and because, in older implementations, the preprocessor was a separate program whose output was passed on to the compiler.

For similar reasons, tokens separated only by a comment may also be combined to form a single token. Here is a summary of how tokenization of a program compiled in **classic** mode is performed:

1. At a given point in the source file, the next token is the longest sequence of characters that can possibly form a token. For example, `i+++++j` is tokenized as `i ++ ++ + j` even though `i ++ + ++ j` may have resulted in a correct program.
2. If the token formed is an identifier and a macro name, the macro is replaced by the text of the tokens specified on its `#define` directive. Each parameter is

replaced by the text of the corresponding argument. Comments are removed from both the arguments and the macro text.

3. Scanning is resumed at the first step from the point at which the macro was replaced, as if it were part of the original program.
4. When the entire program has been preprocessed, the result is scanned again by the compiler as in the first step. The second and third steps do not apply here since there will be no macros to replace. Constructs generated by the first three steps that resemble preprocessing directives are not processed as such.

It is in the third and fourth steps that the text of adjacent but previously separate tokens may be combined to form new tokens.

The \ character for line continuation is accepted only in string and character literals and on preprocessing directives.

Constructs such as:

```
#if 0
  "unterminated
#endif
#define US "Unterminating string
char *s = US terminated now"
```

will not generate diagnostic messages, since the first is an unterminated literal in a FALSE block, and the second is completed after macro expansion. However:

```
char *s = US;
```

will generate a diagnostic message since the string literal in US is not completed before the end of the line.

Empty character literals are allowed. The value of the literal is zero.

Preprocessing directives

The # token must appear in the first column of the line. The token immediately following # is available for macro expansion. The line can be continued with \ only if the name of the directive and, in the following example, the (has been seen:

```
#define f(a,b) a+b
f\
(1,2)      /* accepted */
```

```
#define f(a,b) a+b
f(\
1,2)       /* not accepted */
```

The rules concerning \ apply whether or not the directive is valid. For example,

```
#\
define M 1   /* not allowed */
```

```
#def\
ine M 1      /* not allowed */
```

```
#define\
M 1          /* allowed */
```

```
#dfine\
M 1          /* equivalent to #dfine M 1, even
                  though #dfine is not valid  */
```

Following are the preprocessor directive differences.

#ifdef/#ifndef
When the first token is not an identifier, no diagnostic message is generated, and the condition is FALSE.

#else When there are extra tokens, no diagnostic message is generated.

#endif
When there are extra tokens, no diagnostic message is generated.

#include
The < and > are separate tokens. The header is formed by combining the spelling of the < and > with the tokens between them. Therefore /* and // are recognized as comments (and are always stripped), and the " and ' do begin literals within the < and >. (Remember that in C programs, C++-style comments // are recognized when **-qcpluscmt** is specified.)

#line The spelling of all tokens which are not part of the line number form the new file name. These tokens need not be string literals.

#error
Not recognized.

#define
A valid macro parameter list consists of zero or more identifiers each separated by commas. The commas are ignored and the parameter list is constructed as if they were not specified. The parameter names need not be unique. If there is a conflict, the last name specified is recognized.

For an invalid parameter list, a warning is issued. If a macro name is redefined with a new definition, a warning will be issued and the new definition used.

#undef
When there are extra tokens, no diagnostic message is generated.

Macro expansion

- When the number of arguments on a macro invocation does not match the number of parameters, a warning is issued.
- If the (token is present after the macro name of a function-like macro, it is treated as too few arguments (as above) and a warning is issued.
- Parameters are replaced in string literals and character literals.
- Examples:

```
#define M()    1
#define N(a)   (a)
#define O(a,b) ((a) + (b))
```

```
M();  /* no error */
N();  /* empty argument */
O();  /* empty first argument
            and too few arguments */
```

Text output

No text is generated to replace comments.

-qlargepage

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Takes advantage of large pages provided on POWER4 and higher systems, for applications designed to execute in a large page memory environment.

When **-qlargepage** is in effect to compile a program designed for a large page environment, an increase in performance can occur.

Syntax

```
            ┌─nolargepage─┐
►►── -q─────┴─largepage───┴──────────────────────────────────────────────►◄
```

Defaults

-qnolargepage

Usage

Note that this option is only useful in the following conditions:

- Large pages must be available and configured on the system.
- You must compile with an option that enables loop optimization, such as **-O3** or **-qhot**.
- You must link with the **-blpdata** option.

See your AIX operating system documentation for more information on using large page support.

Predefined macros

None.

Examples

To compile `myprogram.c` to use large page heaps, enter:

```
xlc myprogram.c -qlargepage -blpdata
```

-qldbl128, -qlongdouble

Category

Floating-point and integer control

Pragma equivalent

#pragma options [no]ldbl128

Purpose

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

Syntax

```
           ┌─nolongdouble─┐
           ├─noldbl128────┤
►►── -q────┼─ldbl128──────┼──────────────────────────────────────────────►◄
           └─longdouble───┘
```

Defaults

-qnoldbl128

Usage

Separate libraries are provided that support 128-bit `long double` types. These libraries will be automatically linked if you use any of the invocation commands with the **128** suffix (**xlc128**, **cc128**, **xlc128_r,** or **cc128_r**). You can also manually link to the 128-bit versions of the libraries using the **-l***key* option, as shown in the following table:

Default (64-bit) long double		**128-bit long double**	
Library	**Form of the -l*key* option**	**Library**	**Form of the -l*key* option**
libC.a	-lC	libC128.a	-lC128
libC_r.a	-lC_r	libC128_r.a	-lC128_r

Linking without the 128-bit versions of the libraries when your program uses 128-bit long doubles (for example, if you specify **-qldbl128** alone) may produce unpredictable results.

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

Predefined macros

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

Examples

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

```
xlc myprogram.c -qldbl128 -lC128
```

Related information

- "-l" on page 204

-qlib

Category

Linking

Pragma equivalent

None.

Purpose

Specifies whether standard system libraries and XL C libraries are to be linked.

When **-qlib** is in effect, the standard system libraries and compiler libraries are automatically linked. When **-qnolib** is in effect, the standard system libraries and compiler libraries are not used at link time; only the libraries specified on the command line with the **-l** flag will be linked.

This option can be used in system programming to disable the automatic linking of unneeded libraries.

Syntax

```
            ┌─lib───┐
►►── -q─────┴─nolib─┴──────────────────────────────────────────────►◄
```

Defaults

-qlib

Usage

Using **-qnolib** specifies that no libraries, including the system libraries as well as the XL C libraries (these are found in the lib/aix61 subdirectories of the compiler installation directory), are to be linked. The system startup files are still linked, unless **-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 **-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 -qnolib -lxlopt
```

Related information

- "-qcrt" on page 124

-qlibansi

Category

Optimization and tuning

Pragma equivalent

#pragma options [no]libansi

Purpose

Assumes that all functions with the name of an ANSI C library function are in fact the system functions.

When **libansi** is in effect, the optimizer can generate better code because it will know about the behavior of a given function, such as whether or not it has any side effects.

Syntax

```
           ┌─nolibansi─┐
►►── -q────┴─libansi───┴────────────────────────────────────────────►◄
```

Defaults

-qnolibansi

Predefined macros

None.

-qlibmpi

Category

"Optimization and tuning" on page 85

Pragma equivalent

None

Purpose

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

Syntax

```
           ┌─nolibmpi─┐
►►── -q────┴─libmpi───┴─────────────────────────────────────────────►◄
```

Defaults

-qnolibmpi

Usage

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

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

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

Note: You cannot use this option if your application contains your own version of the library function that is incompatible with the standard one.

Predefined macros

None.

Examples

To compile `myprogram.c`, enter the following command:

```
xlc -O5 myprogram.c -qlibmpi
```

Related information

- Message Passing Interface Forum
- "-qipa" on page 193

-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

```
           ┌─nolinedebug─┐
►►── -q────┴─linedebug───┴──────────────────────────────────────►◄
```

Defaults

-qnolinedebug

Usage

When **-qlinedebug** is in effect, function inlining is disabled.

Avoid using **-qlinedebug** with **-O** (optimization) option. The information produced may be incomplete or misleading.

The **-g** option overrides the **-qlinedebug** option. If you specify **-g** with **-qnolinedebug** on the command line, **-qnolinedebug** is ignored and a warning is issued.

Predefined macros

None.

Examples

To compile `myprogram.c` to produce an executable program `testing` so you can step through it with a debugger, enter:

```
xlc myprogram.c -o testing -qlinedebug
```

Related information

- "-g" on page 160
- "-O, -qoptimize" on page 236

-qlist

Category

Listings, messages, and compiler information

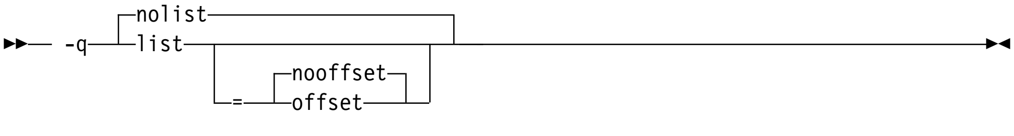
Pragma equivalent

#pragma options [no]list

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 19.

You can use the object or assembly listing to help understand the performance characteristics of the generated code and to diagnose execution problems.

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

Predefined macros

None.

Examples

To compile `myprogram.c` and to produce a listing (.lst) file that includes object , enter:

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

Related information

- "-qlistopt" on page 221
- "-qprint" on page 259
- "-qsource" on page 286

-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

```
                   ┌─xml──┐
►►── -q──listfmt=──┴─html─┴──┬─────────────────────────────────────────┬──────►◄
                             │        ┌─:──────────────────────────┐   │
                             │        ▼                            │   │
                             └─=──────┬─contentSelectionList───────┬┴───┘
                                      ├─filename=──filename────────┤
                                      ├─version=──version number───┤
                                      └─stylesheet=──filename──────┘
```

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 221.

`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 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 for AIX, 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 **-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 -qinline -qlistfmt=xml=inlines myprogram.c
```

genhtml command

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

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

```
genhtml xml_file
```

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

```
genhtml xml_file > target_html_file
```

Note: The 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 263
- "Using compiler reports to diagnose optimization opportunities" in the *XL C Optimization and Programming Guide*

-qlistopt

Category

Listings, messages, and compiler information

Pragma equivalent

None.

Purpose

Produces a compiler listing file that includes all options in effect at the time of compiler invocation.

When **listopt** is in effect, a listing file is generated with a .lst suffix for each source file named on the command line. The listing shows options in effect as set by the compiler defaults, the configuration file, and command line settings. For details of the contents of the listing file, see “Compiler listings” on page 19.

Syntax

```
            ┌─nolistopt─┐
►►── -q─────┴─listopt───┴──────────────────────────────────────────────►◄
```

Defaults

-qnolistopt

Usage

Option settings caused by pragma statements in the program source are not shown in the compiler listing.

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

Predefined macros

None.

Examples

To compile `myprogram.c` to produce a listing (.lst) file that shows all options in effect, enter:

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

Related information

- “-qlist” on page 217
- “-qprint” on page 259
- “-qsource” on page 286

-qlonglit

Category

Floating-point and integer control

Pragma equivalent

None.

Purpose

In 64-bit mode, when determining the implicit types for integer literals, the compiler behaves as if an `l` or `L` suffix were added to integral literals with no suffix or with a suffix consisting only of `u` or `U`.

Syntax

```
          ┌─nolonglit─┐
►►── -q───┴─longlit───┴──────────────────────────────────────────────────────►◄
```

Defaults

-qnolonglit

Usage

After you specify the **-qlonglit** option, if the `int` or `unsigned int` type is contained in the implicit type list of a integer literal, the `int` or `unsigned int` type is replaced with the `long int` or `unsigned long int` type, respectively. For more information about the integer literals, see "Integer literals".

Predefined macros

None.

Examples

After you specify the **-qlonglit** option, the integer literal 0x80000000 has the `long int` type in 64-bit mode. Otherwise, if this option is not specified, the integer literal has the `unsigned int` type in both 32-bit and 64-bit modes.

-qlonglong

Category

Language element control

Pragma equivalent

#pragma options [no]longlong

Purpose

Allows IBM `long long` integer types in your program.

Syntax

```
►►── -q───┬─longlong───┬─────────────────────────────────────────────────────►◄
          └─nolonglong─┘
```

Defaults

- **-qlonglong** for the xlc, cc and c99 invocation commands; **-qnolonglong** for the c89 invocation command.

Usage

This option takes effect when the **-qlanglvl=extended | stdc89 | extc89** option is in effect. It is not valid when the **-qlanglvl=stdc99 | extc99** option is in effect, because the `long long` support provided by this option is incompatible with the semantics of the `long long` types mandated by the C99 standard.

Predefined macros

_LONG_LONG is defined to 1 when `long long` data types are available; otherwise, it is undefined.

Examples

To compile `myprogram.c` with support for IBM `long long` integers, enter the following command:

```
cc myprogram.c -qlonglong
```

AIX v4.2 and later provides support for files greater than 2 gigabytes in size so you can store large quantities of data in a single file. To allow large file manipulation in your application, compile with the -D_LARGE_FILES and **-qlonglong** compiler options. See the following example:

```
xlc myprogram.c -D_LARGE_FILES -qlonglong
```

Related information

- "Integral types" in the *IBM XL C for AIX, V13.1.3 Language Reference*

-ma

See “-qalloca, -ma” on page 100.

-qmacpstr

Category

Language element control

Pragma equivalent

#pragma options [no]macpstr

Purpose

Converts Pascal string literals (prefixed by the `\p` escape sequence) into null-terminated strings in which the first byte contains the length of the string.

For example, when the **-qmacpstr** option is in effect, the compiler converts:

```
"\pABC"
```

to:

```
'\03' , 'A' , 'B' , 'C' , '\0'
```

Syntax

```
           ┌─nomacpstr─┐
►►── -q────┴─macpstr───┴──────────────────────────────────────────────►◄
```

Defaults

-qnomacpstr

Usage

A Pascal string literal always contains the characters `"\p`. The characters `\p` in the middle of a string do not form a Pascal string literal, and must be *immediately preceded* by the " (double quote) character.

Entering the characters:

```
'\p' , 'A' , 'B' , 'C' , '\0'
```

into a character array does not form a Pascal string literal.

The compiler ignores the **-qmacpstr** option when the **-qmbcs** or **-qdbcs** option is active because Pascal-string-literal processing is only valid for one-byte characters.

The **#pragma options** keyword **macpstr** is only valid at the top of a source file before any C source statements. If you attempt to use it in the middle of a source file, it is ignored and the compiler issues an error message.

The following describes how Pascal string literals are processed.

- Because there is no Pascal-string-literal processing of wide strings, using the escape sequence `\p` in a wide string literal with the **-qmacpstr** option, generates a warning message and the escape sequence is ignored.
- Concatenating a Pascal string literal to a normal string gives a non-Pascal string. For example, concatenating the strings:

  ```
  "ABC" "\pDEF"
  ```

  gives:

  ```
  "ABCpDEF"
  ```

- Concatenating two Pascal string literals, for example, `strcat`, does not result in a Pascal string literal. However, as described above, two adjacent Pascal string literals can be concatenated to form one Pascal string literal in which the first byte is the length of the new string literal. For example, concatenating the strings:

  ```
  "\p ABC" "\p DEF"
  ```

  or

  ```
  "\p ABC" "DEF"
  ```

  results in:

  ```
  "\06ABCDEF"
  ```

- A Pascal string literal cannot be concatenated with a wide string literal.
- The compiler truncates a Pascal string literal that is longer than 255 bytes (excluding the length byte and the terminating NULL) to 255 characters.
- The Pascal string literal is *not* a basic type different from other C string literals. After the processing of the Pascal string literal is complete, the resulting string is treated the same as all other strings. If the program passes a C string to a function that expects a Pascal string, or vice versa, the behavior is undefined.
- Modifying any byte of the Pascal string literal after the processing has been completed does not alter the original length value in the first byte. For example, in the string `"\06ABCDEF"`, substituting a null character for one of the existing characters in the middle of the string does not change the value of the first byte of the string, which contains the length of the string.
- No errors or warnings are issued when the bytes of the processed Pascal string literal are modified.

Predefined macros

None.

Examples

To compile mypascal.c and convert string literals into Pascal-style strings, enter:

```
xlc mypascal.c -qmacpstr
```

Related information

- “-qmbcs, -qdbcs” on page 230

-qmakedep, -M

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 .u suffix.

Syntax

```
►►──┬─-M──────────────────────┬──────────────────────────────────►◄
    └─-q──makedep──┬─────────┬┘
                   └─=──gcc──┘
```

Defaults

Not applicable.

Parameters

gcc (-qmakedep option only)
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.

If you specify -**qmakedep** with no suboption, or **-M**, 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 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 .u 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
```

You can also use **-qmakedep** and **-M** with the following option:

-MF *file_path*
Sets the name of the dependency output file, where *file_path* is the full or partial path or file name for the dependency output file. For more information, see "-MF" on page 231.

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 12. If the include file is not found, it is not added to the .u 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.u`, enter:

```
xlc -c -qmakedep mysource.c
```

Example 2: To compile `foo_src.c` and create a dependency output file named `mysource.u`, enter:

```
xlc -c -qmakedep foo_src.c -MF mysource.u
```

Example 3: To compile `foo_src.c` and create a dependency output file named `mysource.u` in the `deps/` directory, enter:

```
xlc -c -qmakedep foo_src.c -MF deps/mysource.u
```

Example 4: To compile `foo_src.c` and create an object file named `foo_obj.o` and a dependency output file named `foo_obj.u`, 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.u`, enter:

```
xlc -c -qmakedep foo_src.c -o foo_obj.o -MF mysource.u
```

Example 6: To compile `foo_src1.c` and `foo_src2.c` to create two dependency output files, named `foo_src1.u` and `foo_src2.u` respectively, in the `/tmp/` directory, enter:

```
xlc -c -qmakedep foo_src1.c foo_src2.c -MF /tmp/
```

Related information

- "-MF" on page 231
- "-o" on page 235
- "Directory search sequence for included files" on page 12

-qmaxerr

Category

Error checking and debugging

Pragma equivalent

None.

Purpose

Stops compilation when the number of error messages of a specified severity level or higher reaches a specified number.

Syntax

-qmaxerr syntax — C

```
        ┌─nomaxerr──────────────────────────┐
►►── -q─┴─maxerr──=──number─┬────────────┬──┴──────────────────────────►◄
                            │    ┌─s─┐   │
                            └─:──┼─i─┼───┘
                                 ├─w─┤
                                 └─e─┘
```

Defaults

-qnomaxerr

Parameters

number
It specifies the maximum number of messages the compiler generates before it stops. *number* must be an integer with a value of 1 or greater.

i Specifies that the severity level is Informational (I) or higher.

w Specifies that the severity level is Warning (W) or higher.

e Specifies that the severity level is Error (E) or higher.

s Specifies that the severity level is Severe (S).

Usage

If the **-qmaxerr** option does not specify the severity level, it uses the severity that is in effect by the **-qhalt** option; otherwise, the severity level is specified by either **-qmaxerr** or **-qhalt** that appears last.

Diagnostic messages can be controlled by the **-qflag** option.

Predefined macros

None.

Examples

To stop compilation of `myprogram.c` when 10 warnings are encountered, enter the command:

```
xlc myprogram.c -qmaxerr=10:w
```

To stop compilation of `myprogram.c` when 5 severe errors are encountered, assuming that the current **-qhalt** option value is **s** (severe), enter the command:

```
xlc myprogram.c -qmaxerr=5
```

To stop compilation of `myprogram.c` when 3 informational messages are encountered, enter the command:

```
xlc myprogram.c -qmaxerr=3:i
```

or:

```
xlc myprogram.c -qmaxerr=3 -qhalt=i
```

Related information

- "-qflag" on page 145
- "-qhalt" on page 165
- "Message severity levels and compiler response" on page 17

-qmaxmem

Category

Optimization and tuning

Pragma equivalent

#pragma options maxmem

Purpose

Limits the amount of memory that the compiler allocates while performing specific, memory-intensive optimizations to the specified number of kilobytes.

Syntax

```
►►── -q──maxmem──=──size_limit──────────────────────────────────────►◄
```

Defaults

- **-qmaxmem=8192** when **-O2** is in effect.
- **-qmaxmem=-1** when the **-O3** or higher optimization level is in effect.

Parameters

size_limit
The number of kilobytes worth of memory to be used by optimizations. The limit is the amount of memory for specific optimizations, and not for the

compiler as a whole. Tables required during the entire compilation process are not affected by or included in this limit.

A value of -1 permits each optimization to take as much memory as it needs without checking for limits.

Usage

A smaller limit does not necessarily mean that the resulting program will be slower, only that the compiler may finish before finding all opportunities to increase performance. Increasing the limit does not necessarily mean that the resulting program will be faster, only that the compiler is better able to find opportunities to increase performance if they exist.

Setting a large limit has no negative effect on the compilation of source files when the compiler needs less memory. However, depending on the source file being compiled, the size of subprograms in the source, the machine configuration, and the workload on the system, setting the limit too high, or to -1, might exceed available system resources.

Predefined macros

None.

Examples

To compile `myprogram.c` so that the memory specified for local table is 16384 kilobytes, enter:

```
xlc myprogram.c -qmaxmem=16384
```

-qmbcs, -qdbcs

Category

Language element control

Pragma equivalent

#pragma options [no]mbcs, #pragma options [no]dbcs

Purpose

Enables support for multibyte character sets (MBCS) and Unicode characters in your source code.

When **mbcs** or **dbcs** is in effect, multibyte character literals and comments are recognized by the compiler. When **nombcs** or **nodbcs** is in effect, the compiler treats all literals as single-byte literals.

Syntax

```
           ┌─nodbcs─┐
           ├─nombcs─┤
►►─ -q─────┼─mbcs───┼──────────────────────────────────────────────►◄
           └─dbcs───┘
```

Defaults

-qnombcs, -qnodbcs

Usage

For rules on using multibyte characters in your source code, see "Multibyte characters" in the *XL C Language Reference*.

In addition, you can use multibyte characters in the following contexts:

- In file names passed as arguments to compiler invocations on the command line; for example:
  ```
  xlc /u/myhome/c_programs/kanji_files/multibyte_char.c -omultibyte_char
  ```
- In file names, as suboptions to compiler options that take file names as arguments
- In the definition of a macro name using the **-D** option; for example:
  ```
  -DMYMACRO="kpsmultibyte_chardcs"
  -DMYMACRO='multibyte_char'
  ```

Listing files display the date and time for the appropriate international language, and multibyte characters in the source file name also appear in the name of the corresponding list file. For example, a C source file called:

```
multibyte_char.c
```

gives a list file called

```
multibyte_char.lst
```

Predefined macros

None.

Examples

To compile `myprogram.c` if it contains multibyte characters, enter:

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

Related information

- "-D" on page 126

-MF

Category

Output control

Pragma equivalent

None.

Purpose

Specifies the name or location for the dependency output files that are generated by the **-qmakedep** or **-M** option.

For more information about the **-qmakedep** and **-M** options, see “-qmakedep, -M” on page 226.

Syntax

```
►►── -MF──file_path──────────────────────────────────────────────►◄
```

Defaults

If **-MF** is not specified, the dependency output file is generated with the same name as the object file but with a .u suffix in the current working directory.

Parameters

file_path
: The target output path. *file_path* can be a full directory path or file name. If *file_path* is the name of a directory, the dependency file generated by the compiler is placed into the specified directory. If you do not specify a directory, the dependency file is stored in the current working directory.

Usage

If the file specified by **-MF** option already exists, it will be overwritten.

If you specify a single file name for the **-MF** option when you compile multiple source files, only a single dependency file will be generated. The dependency file contains the **make** rule for the last file specified on the command line.

Predefined macros

None.

Related information

- “-qmakedep, -M” on page 226
- “-o” on page 235
- “Directory search sequence for included files” on page 12

-qminimaltoc

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Controls the generation of the table of contents (TOC), which the compiler creates for an executable file.

Programs compiled in 64-bit mode have a limit of 8192 TOC entries. As a result, you may encounter "relocation truncation" error messages when linking large programs in 64-bit mode; these error messages are caused by TOC overflow conditions. When **-qminimaltoc** is in effect, the compiler avoids these overflow conditions by placing TOC entries into a separate data section for each object file.

Specifying **-qminimaltoc** ensures that the compiler creates only one TOC entry for each compilation unit. Specifying this option can minimize the use of available TOC entries, but its use impacts performance. Use the **-qminimaltoc** option with discretion, particularly with files that contain frequently executed code.

Syntax

```
            ┌─nominimaltoc─┐
►►── -q─────┴─minimaltoc───┴──────────────────────────────────────────►◄
```

Defaults

-qnominimaltoc

Usage

Compiling with **-qminimaltoc** may create slightly slower and larger code for your program. However, these effects may be minimized by specifying optimizing options when compiling your program.

Predefined macros

None.

-qmkshrobj

Category

Output control

Pragma equivalent

None.

Purpose

Creates a shared object from generated object files.

Use this option, together with the related options described later in this topic, instead of calling the linker directly to create a shared object. The advantage of using this option is that it is compatible with **-qipa** link-time optimizations (such as those performed at **-O5**).

Syntax

```
►►── -q──mkshrobj─────────────────────────────────────────────────────►◄
```

Defaults

By default, the output object is linked with the runtime libraries and startup routines to create an executable file.

Usage

When the **-qmkshrobj** option is specified, the driver program starts the CreateExportList utility to create an export list from the input list of object files.

The compiler automatically exports all global symbols from the shared object unless you specify which symbols to export by using **-bE**:, **-bexport**:, or **-bnoexpall**. You can also prevent weak symbols from being exported by using the **-qnoweakexp** option. ▶ IBM Symbols that have the hidden or internal visibility attribute are not exported. IBM ◀

Specifying **-qmkshrobj** implies **-qpic**.

You can also use the following related options with **-qmkshrobj**:

`-o` *`shared_file`*
: The name of the file that holds the shared file information. The default is shr.o.

`-qexpfile=`*`filename`*
: Saves all exported symbols in *filename*.

`-e` *`name`*
: Sets the entry name for the shared executable to *name*.

`-q[no]weakexp`
: Specifies whether symbols marked as weak (with the **#pragma weak** directive) are to be included in the export list. If you do not explicitly set this option, the default is **-qweakexp** (global weak symbols are exported).

For detailed information about using **-qmkshrobj** to create shared libraries, see "Constructing a library" in the *XL 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 -qmkshrobj -o big_lib.so lib_a.o lib_b.o lib_c.o
```

Related information

- "-b" on page 109
- "-e" on page 135
- "-G" on page 163
- "-qexpfile" on page 141
- "-qipa" on page 193
- "-o" on page 235
- "-qpic" on page 254
- "-qweakexp" on page 328
- "-qvisibility" on page 323
- "#pragma GCC visibility push, #pragma GCC visibility pop" on page 349

-o

Category

Output control

Pragma equivalent

None.

Purpose

Specifies a name for the output object, assembler, executable, or preprocessed file.

Syntax

```
►►— -o—path————————————————————————————————————————►◄
```

Defaults

See “Types of output files” on page 4 for the default file names and suffixes produced by different phases of compilation.

Parameters

path

When you are using the option to compile from source files, *path* can be the name of a file or directory. *path* can be a relative or absolute path name. When you are using the option to link from object files, *path* must be a file name.

If *path* is the name of an existing directory, files created by the compiler are placed into that directory. If *path* is not an existing directory, it specifies the name of the file produced by the compiler. See below for examples.

You cannot specify a file name with a C source file suffix (.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** and *path* is not an existing directory, 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 **-E, -P**, and **-qsyntaxonly** options override the **-o** option.

Predefined macros

None.

Examples

To compile `myprogram.c` so that the resulting executable is called `myaccount`, assuming that no directory with name `myaccount` exists, enter:

```
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 114
- “-E” on page 136
- “-P” on page 244
- “-qsyntaxonly” on page 302

-O, -qoptimize

Category

Optimization and tuning

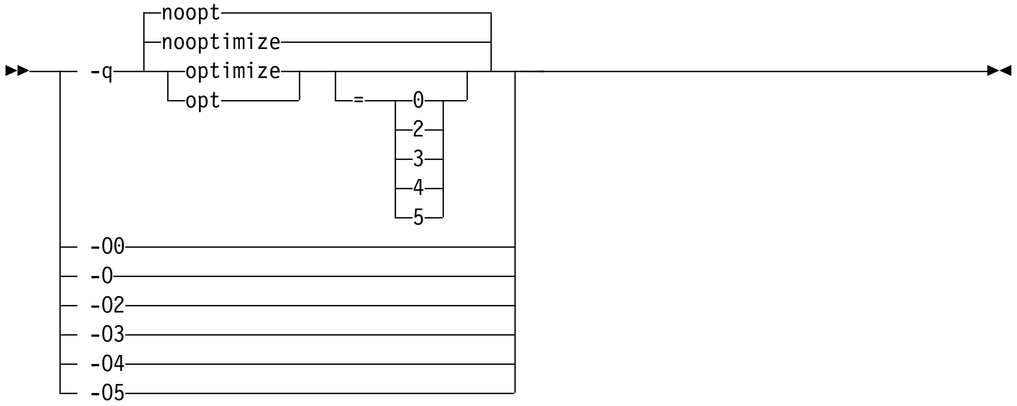
Pragma equivalent

#pragma options [no]optimize

Purpose

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

Syntax



Defaults

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

Parameters

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

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

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

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

`-O3 | optimize|opt=3`

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

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

1. Aggressive code motion, and scheduling on computations that have the potential to raise an exception, are allowed.

   Loads and floating-point computations fall into this category. This optimization is aggressive because it may place such instructions onto execution paths where they *will* be executed when they *may* not have been according to the actual semantics of the program.

   For example, a loop-invariant floating-point computation that is found on some, but not all, paths through a loop will not be moved at **-O2** because the computation may cause an exception. At **-O3**, the compiler will move it because it is not certain to cause an exception. The same is true for motion of loads. Although a load through a pointer is never moved, loads off the static or stack base register are considered movable at **-O3**. Loads in general are not considered to be absolutely safe at **-O2** because a program can contain a declaration of a static array `a` of 10 elements and load `a[60000000003]`, which could cause a segmentation violation.

   The same concepts apply to scheduling.

   Example:

   In the following example, at **-O2**, the computation of `b+c` is not moved out of the loop for two reasons:

   - It is considered dangerous because it is a floating-point operation
   - It does not occur on every path through the loop

   At **-O3**, the code is moved.

   ```
     ...
   int i ;
   float a[100], b, c ;
   for (i = 0 ; i < 100 ; i++)
    {
    if (a[i] < a[i+1])
     a[i] = b + c ;
    }
     ...
   ```

2. 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.

3. Floating-point expressions may be rewritten.

   Computations such as **a*b*c** may be rewritten as **a*c*b** if, for example, an opportunity exists to get a common subexpression by such rearrangement. Replacing a divide with a multiply by the reciprocal is another example of reassociating floating-point computations.
4. Specifying **-O3** implies **-qhot=level=0**, unless you explicitly specify **-qhot** or **-qhot=level=1** option.

-qfloat=fltint: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**.

Aggressive optimizations do *not* include the following floating-point suboptions: **-qfloat=hsflt | hssngl**, or anything else that affects the precision mode of a program.

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

Refer to “-qflttrap” on page 151 to see the behavior of the compiler when you specify **optimize** options with the **-qflttrap** option.

You can use the **-qstrict** and **-qstrict_induction** compiler options to turn off effects of **-O3** that might change the semantics of a program. Specifying **-qstrict** together with **-O3** invokes all the optimizations performed at **-O2** as well as further loop optimizations. Reference to the **-qstrict** compiler option can appear before or after the **-O3** option.

The **-O3** compiler option followed by the **-O** option leaves **-qignerrno** on.

When **-O3** and **-qhot=level=1** are in effect, the compiler replaces any calls in the source code to standard math library functions with calls to the equivalent MASS library functions, and if possible, the vector versions.

-O4 | optimize|opt=4

This option is the same as **-O3**, except that it also:

- Sets the **-qarch** and **-qtune** options to the architecture of the compiling machine
- Sets the **-qcache** option most appropriate to the characteristics of the compiling machine
- Sets the **-qhot** option
- Sets the **-qipa** option

Note: Later settings of **-O**, **-qcache**, **-qhot**, **-qipa**, **-qarch**, and **-qtune** options will override the settings implied by the **-O4** option.

This option follows the "last option wins" conflict resolution rule, so any of the options that are modified by **-O4** can be subsequently changed. For example, specifying **-O4 -qarch=ppc** allows aggressive intraprocedural optimization while maintaining code portability.

-O5 | optimize|opt=5

This option is the same as **-O4**, except that it:

- Sets the **-qipa=level=2** option to perform full interprocedural data flow and alias analysis.

Note: Later settings of **-O**, **-qcache**, **-qipa**, **-qarch**, and **-qtune** options will override the settings implied by the **-O5** option.

Usage

Increasing the level of optimization may or may not result in additional performance improvements, depending on whether additional analysis detects further opportunities for optimization.

Compilations with optimizations may require more time and machine resources than other compilations.

Optimization can cause statements to be moved or deleted, and generally should not be specified along with the **-g** flag for debugging programs. The debugging information produced may not be accurate.

When using **-O** or higher optimization, **-qtbtable=small** is implied. The traceback table generated has no function name or parameter information.

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 169
- “-qipa” on page 193
- “-qpdf1, -qpdf2” on page 247
- “-qstrict” on page 294
- “-qtbtable” on page 305
- "Optimizing your applications" in the *XL C Optimization and Programming Guide*.
- “#pragma option_override” on page 364

-qoptdebug

Category

Error checking and debugging

Pragma equivalent

None.

Purpose

When used with high levels of optimization, produces files containing optimized pseudocode that can be read by a debugger.

An output file with a .optdbg extension is created for each source file compiled with **-qoptdebug**. You can use the information contained in this file to help you understand how your code actually behaves under optimization.

Syntax

```
         ┌─nooptdebug─┐
►►──-q───┴─optdebug───┴──────────────────────────────────────────►◄
```

Defaults

-qnooptdebug

Usage

-qoptdebug only has an effect when used with an option that enables the high-level optimizer, namely **-O3** or higher optimization level, or **-qhot**, **-qsmp,** **-qpdf,** or **-qipa**. You can use the option on both compilation and link steps. If you specify it on the compile step, one output file is generated for each source file. If you specify it on the **-qipa** link step, a single output file is generated.

The naming rules of a .optdbg file are as follows:

- If a .optdbg file is generated at the compile step, its name is based on the output file name of the compile step.
- If a .optdbg file is generated at the link step, its name is based on the output file name of the link step.

If you compile and link in the same step using the **-qoptdebug** option with **-qipa**, the .optdbg file is generated only at the link step.

You must still use the **-g** or **-qlinedebug** option to include debugging information that can be used by a debugger.

For more information and examples of using this option, see "Using -qoptdebug to help debug optimized programs" in the *XL C Optimization and Programming GuideXL C Optimization and Programming Guide*.

Related information

- "-O, -qoptimize" on page 236
- "-qhot" on page 169
- "-qipa" on page 193
- "-qpdf1, -qpdf2" on page 247
- "-qsmp" on page 281
- "-g" on page 160
- "-qlinedebug" on page 216

-qoptfile

Category

Compiler customization

Pragma equivalent

None.

Purpose

Specifies a file containing a list of additional command line options to be used for the compilation.

Syntax

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

Defaults

None.

Parameters

filename
: Specifies the name of the file that contains a list of additional command line options. *filename* can contain a relative path or absolute path, or it can contain no path. It is a plain text file with one or more command line options per line.

Usage

The format of the option file follows these rules:

- Specify the options you want to include in the file with the same syntax as on the command line. The option file is a whitespace-separated list of options. The following special characters indicate whitespace: `\n`, `\v`, `\t`. (All of these characters have the same effect.)
- A character string between a pair of single or double quotation marks are passed to the compiler as one option.
- You can include comments in the options file. Comment lines start with the # character and continue to the end of the line. The compiler ignores comments and empty lines.

When processed, the compiler removes the **-qoptfile** option from the command line, and sequentially inserts the options included in the file before the other subsequent options that you specify.

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

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

Predefined macros

None.

Example 1

This is an example of specifying an option 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
-qpic

$ xlC -qlist -qoptfile=options.file -qipa test.c
```

The preceding example is equivalent to the following invocation:

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

Example 2

This is an example of specifying an option file that contains **-qoptfile** with a cycle.

```
$ cat options.file2
# 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 option file
-qoptfile=options.file2
# To generate position-independent code
-qpic
# To produce a compiler listing file
-qlist

$ xlC -qlist -qoptfile=options.file2 -qipa test.c
```

The preceding example is equivalent to the following invocation:

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

Example 3

This is an example of specifying an option file that contains **-qoptfile** without a cycle.

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

$ cat options.file2
-qchars=signed

$ xlC -qoptfile=options.file1 test.c
```

The preceding example is equivalent to the following invocation:

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

Example 4

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

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

$ xlC -qsaveopt -qipa -qoptfile=options.file3 test.c -c

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

Related information

- "-qsaveopt" on page 272

-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 a program compiled with **-p**, and it ends normally, it writes the recorded information to a mon.out file; a program compiled with **-pg** writes a gmon.out file. You can then use the **prof** or **gprof** command to generate a runtime profile.

Syntax

```
►►──┬─ -p──────────────────────┬────────────────────────────────────────────►◄
    ├─ -pg─────────────────────┤
    └─ -q──profile──=──┬─p──┬──┘
                       └─pg─┘
```

Defaults

Not applicable.

Usage

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

If the **-qtbtable** option is not set, the profiling options will generate full traceback tables.

Predefined macros

None.

Examples

To compile `myprogram.c` to include profiling data, enter:

```
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

- "-qtbtable" on page 305
- See your operating system documentation for more information on the **prof** and **gprof** command.

-P

Category

Output control

Pragma equivalent

None.

Purpose

Preprocesses the source files named in the compiler invocation, without compiling, and creates an output preprocessed file for each input file.

The preprocessed output file has the same name as the input file but with a .i suffix.

Syntax

```
▶▶── -P ──────────────────────────────────────────────────────────▶◀
```

Defaults

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

Usage

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

Unless **-qppline** is specified, `#line` directives are not generated.

Line continuation sequences are removed and the source lines are concatenated.

The **-P** option retains all white space including line-feed characters, with the following exceptions:

- All comments are reduced to a single space (unless **-C** is specified).
- Line feeds at the end of preprocessing directives are not retained.
- White space surrounding arguments to function-style macros is not retained.

The **-P** option is overridden by the **-E** option. The **-P** option overrides the **-c**, **-o**, and **-qsyntaxonly** option.

Predefined macros

None.

Related information

- "-C, -C!" on page 115
- "-E" on page 136
- "-qppline" on page 255
- "-qsyntaxonly" on page 302

-qpath

Category

Compiler customization

Pragma equivalent

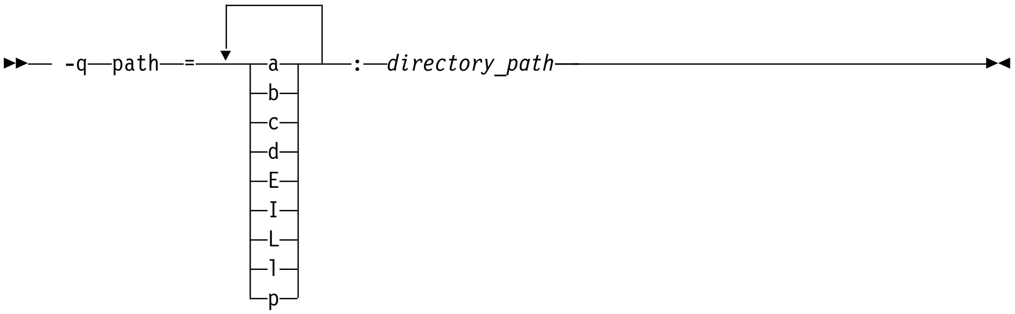
None.

Purpose

Specifies substitute path names for XL 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 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	The compiler front end	xlcentry
d	The disassembler	dis
E	The CreateExportList utility	CreateExportList
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.

Note that using the **p** suboption causes 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` 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 110
- "-F" on page 143
- "-t" on page 303

-qpdf1, -qpdf2

Category

Optimization and tuning

Pragma equivalent

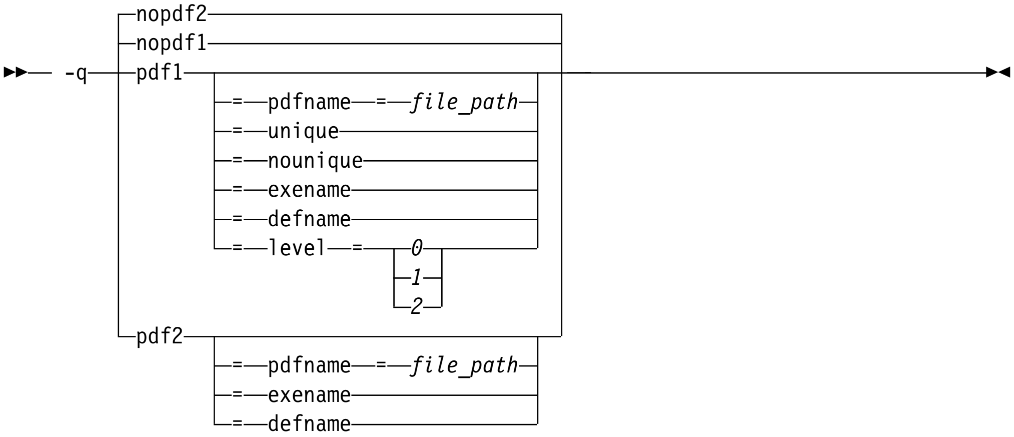
None.

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 24. Profiling type supported on each -qpdf1 level

Profiling type	Level		
	0	**1**	**2**
Block-counter profiling	+	+	+
Call-counter profiling	+	+	+
Value profiling		+	+
Cache-miss profiling			+

-qpdf1=level=1 is the default level. It is equivalent to **-qpdf1**. Higher PDF levels profile more optimization opportunities but have a larger overhead.

Notes:
- Only one application compiled with the **-qpdf1=level=2** option can be run at a time on a particular processor.
- Cache-miss profiling is enabled on **pSeries** system, and is only available on POWER5 processors or higher.
- 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 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 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 277 for more information.

For recommended procedures of using PDF, see "Using profile-directed feedback" in the *XL 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

```
►►──cleanpdf──┬────────┬──┬────┬──┬──────────────┬────────────────────────►◄
              └─pdfdir─┘  └─-u─┘  └─-f──pdfname─┘
```

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

```
              ┌──────────────────────────────┐
              ▼                              │
►►──mergepdf──┬───────────────┬──input──┴── -o──output──┬────┬──┬────┬──────►◄
              └─ -r──scaling─┘                          └─ -n─┘  └─ -v─┘
```

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 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 277
- “-qipa” on page 193
- -qprefetch
- “-qreport” on page 263
- "Optimizing your applications" in the *XL C Optimization and Programming Guide*
- “Runtime environment variables” on page 24
- "Profile-directed feedback" in the *XL C Optimization and Programming Guide*

-qphsinfo

Category

Listings, messages, and compiler information

Pragma equivalent

None.

Purpose

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

Syntax

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

Defaults

-qnophsinfo

Usage

The output takes the form *number1*/*number2* for each phase where *number1* represents the CPU time used by the compiler and *number2* represents real time (wall clock time).

The time reported by `-qphsinfo` is in seconds.

Predefined macros

None.

Examples

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

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

The output will look similar to:

```
C Init    - Phase Ends;   0.010/  0.040
IL Gen    - Phase Ends;   0.040/  0.070
W-TRANS   - Phase Ends;   0.000/  0.010
OPTIMIZ   - Phase Ends;   0.000/  0.000
REGALLO   - Phase Ends;   0.000/  0.000
AS        - Phase Ends;   0.000/  0.000
```

Compiling the same program with **-O4** gives:

```
C Init    - Phase Ends;   0.010/  0.040
IL Gen    - Phase Ends;   0.060/  0.070
IPA       - Phase Ends;   0.060/  0.070
IPA       - Phase Ends;   0.070/  0.110
W-TRANS   - Phase Ends;   0.060/  0.180
OPTIMIZ   - Phase Ends;   0.010/  0.010
REGALLO   - Phase Ends;   0.010/  0.020
AS        - Phase Ends;   0.000/  0.000
```

Compiling the same program with **-O4** gives:

```
Front End - Phase Ends;   0.004/  0.006
IPA       - Phase Ends;   0.040/  0.040
IPA       - Phase Ends;   0.220/  0.280
W-TRANS   - Phase Ends;   0.030/  0.110
OPTIMIZ   - Phase Ends;   0.030/  0.030
REGALLO   - Phase Ends;   0.010/  0.050
AS        - Phase Ends;   0.000/  0.000
```

-qpic

Category

Object code control

Pragma equivalent

None.

Purpose

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

Syntax

```
►►── -q──pic──┬──────────────────┬──────────────────────────────────────────►◄
              │      ┌─small─┐   │
              └─=────┴─large─┴───┘
```

Defaults

- **-qpic=small**

Specifying **-qpic** without any suboptions is equivalent to **-qpic=small**.

Parameters

`small`
Instructs the compiler to assume that the size of the Table of Contents (TOC) is no larger than 64 Kb. When **-qpic=small** is in effect, the compiler generates one instruction for each TOC access.

`large`
Instructs the compiler to assume that the size of the TOC is larger than 64 Kb. When **-qpic=large** is in effect, the compiler generates two instructions for each TOC access to enlarge the accessing range. This helps avoid TOC overflow conditions when the Table of Contents is larger than 64 Kb.

Usage

You must specify **-qpic** or **-qpic=large** when you build shared libraries.

Specifying **-qpic=large** has the same effect as passing **-bbigtoc** to **ld**.

You can use different TOC access options for different compilation units in an application.

Note: For applications whose TOC size is larger than 64K, using **-qpic=large** can improve performance. However, for applications whose TOC is smaller than 64K, using **-qpic=large** slows down the program. To decide whether to use **-qpic=large**, compile the program with **-qpic=small** first. If an overflow error message is generated, use **-qpic=large** instead.

Note: If your operating system is lower than AIX 6.1 TL 6, ensure that you have installed the latest fix pack from https://www.ibm.com/support/docview.wss?uid=isg1fixinfo118013; otherwise, an error message might be generated during the link step.

Predefined macros

None.

Examples

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

```
xlc mylib.c -qpic=small -c -o mylib.o
xlc -qmkshrobj mylib -o libmylib.so.1
```

Related information

- “-q32, -q64” on page 94
- “-G” on page 163
- “-qmkshrobj” on page 233

-qppline

Category

Object code control

Pragma equivalent

None.

Purpose

When used in conjunction with the **-E** or **-P** options, enables or disables the generation of `#line` directives.

Syntax

```
►►— -q—┬—ppline——┬——————————————————————————————————————————————————————►◄
       └—noppline—┘
```

Defaults

- **-qnoppline** when **-P** is in effect
- **-qppline** when **-E** is in effect

Usage

The **-C** option has no effect without either the **-E** or the **-P** option. With the **-E** option, line directives are written to standard output. With the **-P** option, line directives are written to an output file.

Predefined macros

None.

Examples

To preprocess `myprogram.c` to write the output to `myprogram.i`, and generate `#line` directives:

```
xlc myprogram.c -P -qppline
```

Related information

- "-E" on page 136
- "-P" on page 244

-qprefetch

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Inserts prefetch instructions automatically where there are opportunities to improve code performance.

When **-qprefetch** is in effect, the compiler may insert prefetch instructions in compiled code. When **-qnoprefetch** is in effect, prefetch instructions are not inserted in compiled code.

Syntax

```
                   ┌─:─────────────────────────────────────┐
                   ↓                                       │
         ┌─prefetch──┬─────────────────────────────────────┬┴─┐
         │           │      ┌─noassistthread──────────┐    │  │
         │           ├─=──┴─assistthread──=──┬─SMT─┬─┴────┤  │
         │           │                        └─CMP─┘      │  │
         │           │      ┌─noaggressive─┐               │  │
         │           ├─=──┴─aggressive─────┴──────────────┤  │
         │           └─=──dscr──=──value───────────────────┘  │
►►── -q──┴─noprefetch─────────────────────────────────────────┴──────────►◄
```

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 value to control the hardware prefetch engine. For POWER8 processors, the value is valid only when the optimization level is `-O2` or greater; for POWER5, POWER6, and POWER7 processors, the value is valid only when the optimization level is `-O3` or greater and the high-order transformation (HOT) is in effect. 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 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

- -qarch
- “-qhot” on page 169
- “-qpdf1, -qpdf2” on page 247
- “-qreport” on page 263
- “__mem_delay” on page 611

-qprint

Category

Listings, messages, and compiler information

Pragma equivalent

None.

Purpose

Enables or suppresses listings.

When **-qprint** is in effect, listings are enabled if they are requested by other compiler options that produce listings. When **-qnoprint** is in effect, all listings are suppressed, regardless of whether listing-producing options are specified.

Syntax

```
           ┌─print───┐
►►── -q────┴─noprint─┴──────────────────────────────────────────────►◄
```

Defaults

-qprint

Usage

You can use **-qnoprint** to override all listing-producing options and equivalent pragmas, regardless of where they are specified. These options are:

- -qattr
- -qlist

- -qlistopt
- -qsource
- -qxref

Predefined macros

None.

Examples

To compile `myprogram.c` and suppress all listings, even if some files have **#pragma options source** and similar directives, enter:

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

-qprocimported, -qproclocal, -qprocunknown

Category

Optimization and tuning

Pragma equivalent

#pragma options proclocal, #pragma options procimported, #pragma options procunknown

Purpose

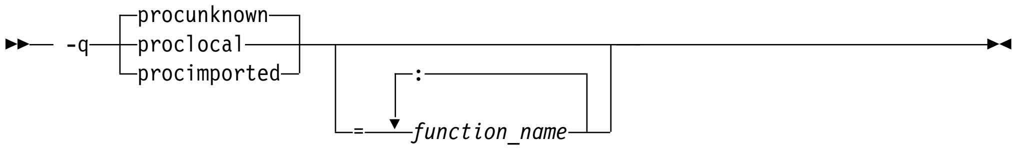
Marks functions as local, imported, or unknown.

Local functions are statically bound with the functions that call them; smaller, faster code is generated for calls to such functions. You can use the **-qproclocal** option or pragma to name functions that the compiler can assume to be local.

Imported functions are dynamically bound with a shared portion of a library. Code generated for calls to functions marked as imported may be larger, but is faster than the default code sequence generated for functions marked as unknown. You can use the **-qprocimported** option or pragma to name functions that the compiler can assume to be imported.

Unknown functions are resolved to either statically or dynamically bound objects during linking. You can use the **-qprocunkown** option or pragma to name functions that the compiler can assume to be unknown.

Syntax



Defaults

-qprocunkown: The compiler assumes that all functions' definitions are unknown.

Parameters

function_name
The name of a function that the compiler should assume to be local, imported, or unknown (depending on the option specified). If you do not specify any *function_name*, the compiler assumes that *all* functions are local, imported, or unknown.

Usage

If any functions that are marked as local resolve to shared library functions, the linker will detect the error and issue warnings. If any of the functions that are marked as imported resolve to statically bound objects, the generated code may be larger and run more slowly than the default code sequence generated for unknown functions.

If a function satisfies all of the following conditions, the compiler issues a warning message to indicate that the final executable file might have a performance loss:

- Has a local definition.
- Is marked as imported or unknown.
- ▶ IBM Has the protected, hidden, or internal visibility attribute. IBM ◀

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

Predefined macros

None.

Examples

To compile `myprogram.c` along with the archive library `oldprogs.a` so that:

- Functions `fun` and `sun` are specified as local
- Functions `moon` and `stars` are specified as imported
- Function `venus` is specified as unknown

use the following command:

```
xlc myprogram.c oldprogs.a -qprolocal=fun(int):sun()
  -qprocimported=moon():stars(float) -qprocunknown=venus()
```

If the following example, in which a function marked as local instead resolves to a shared library function, is compiled with **-qproclocal**:

```
int main(void)
{
    printf("Just in function foo1()\n");
    printf("Just in function foo1()\n");
}
```

a linker error will result. To correct this problem, you should explicitly mark the called routine as being imported from a shared object. In this case, you would recompile the source file and explicitly mark `printf` as imported by compiling with `-qproclocal -qprocimported=printf`.

Related information

- “-qdataimported, -qdatalocal, -qtocdata” on page 127
- “-qvisibility” on page 323
- “#pragma GCC visibility push, #pragma GCC visibility pop” on page 349

-qproto

Category

Object code control

Pragma equivalent

#pragma options [no]proto

Purpose

Specifies the linkage conventions for passing floating-point arguments to functions that have not been prototyped.

When **proto** is in effect, the compiler assumes that the arguments in function calls are the same types as the corresponding parameters of the function definition, even if the function has not been prototyped. By asserting that an unprototyped function actually expects a floating-point argument if it is called with one, you allow the compiler to pass floating-point arguments in floating-point registers exclusively. When **noproto** is in effect, the compiler does not make this assumption, and must pass floating-point parameters in floating-point and general purpose registers.

Syntax

```
           ┌─noproto─┐
►►── -q────┴─proto───┴───────────────────────────────────────────►◄
```

Defaults

-qnoproto

Usage

This option is only valid when the compiler allows unprototyped functions; that is, with the **cc** or **xlc** invocation command, or with the **-qlanglvl** option set to **classic | extended | extc89 | extc99**.

Predefined macros

None.

Examples

To compile `my_c_program.c` to allow the compiler to use the standard linkage conventions for floating-point parameters, even when functions are not prototyped, enter:

```
xlc my_c_program.c -qproto
```

-r

Category

Object code control

Pragma equivalent

None.

Purpose

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

Syntax

```
►►— -r————————————————————————————————————————————————————————►◄
```

Defaults

Not applicable.

Usage

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

Predefined macros

None.

Examples

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

```
xlc myprogram.c myprog2.c -r -o mytest.o
```

Related information

- -qipa

-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 automatic parallelization or vectorization, the listing file shows a pseudo-C code listing and a summary of how program loops are parallelized or optimized. The report also includes diagnostic information about why specific loops cannot be parallelized or 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. POWER4 and POWER5 processors support load stream prefetch and POWER6 or higher processors support both load and store stream prefetch.

Syntax

```
             ┌─noreport─┐
►►── -q──────┴─report───┴─────────────────────────────────────────────►◄
```

Defaults

-qnoreport

Usage

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

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

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
```

To compile `myprogram.c` so the compiler listing also includes a report showing how parallelized loops are transformed, enter:

```
xlc_r -qhot -qsmp -qreport myprogram.c
```

Related information

- “-qhot” on page 169
- “-qsimd” on page 278
- “-qipa” on page 193
- “-qsmp” on page 281
- “-qoptdebug” on page 239
- “-qprefetch” on page 256
- "Using -qoptdebug to help debug optimized programs" in the *XL C Optimization and Programming Guide*

-qreserved_reg

Category

Object code control

Pragma equivalent

None.

Purpose

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

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

Syntax

```
                          ┌─:─────────────┐
                          ▼               │
►►── -q──reserved_reg──=───register_name──┴──────────────────────────►◄
```

Defaults

Not applicable.

Parameters

register_name
A valid register name on the target platform. Valid registers 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 the specified functions, except that you do not need to modify the source file.

Syntax

```
         ┌─norestrict─┐
►►──-q───┴─restrict───┴──┬───────────────────────────┬──────────────────────────►◄
                         │       ┌─:─────────────┐   │
                         │       ▼               │   │
                         └─=─────function_name───┴───┘
```

Defaults

-qnorestrict. It means no function pointer parameters are restricted, unless you specify the **restrict** attribute in the source file.

Usage

If you do not specify the *function_name*, pointer parameters in all functions are treated as **restrict**. Otherwise, only those pointer parameters in the listed functions are treated as **restrict**.

function_name is a colon-separated list.

Using this option can improve the performance of your application, but incorrectly asserting this pointer restriction might cause the compiler to generate incorrect code based on the false assumption. If the application works correctly when recompiled without **-qrestrict**, the assertion might be false. In this case, this option should not be used.

Notes:

- Using **-qnokeyword=restrict** has no impact on the **-qrestrict** option.
- If you 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

- The restrict type qualifier in the *XL C Language Reference*.
- Keywords in the *XL C Language Reference*.
- "-qkeyword" on page 203
- "-qalias" on page 96

-qro

Category

Object code control

Pragma equivalent

#pragma options ro, #pragma strings

Purpose

Specifies the storage type for string literals.

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

Syntax

Option syntax

```
          ┌ro──┐
►►── -q───┴noro┴──────────────────────────────────────────────────────►◄
```

Pragma syntax

```
                                ┌readonly──┐
►►──#──pragma──strings──(───┴writeable─┴───)─────────────────────────►◄
```

Defaults

Strings are read-only for all invocation commands except **cc**. If the **cc** invocation command is used, strings are writeable.

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 267
- "-qroconst"

-qroconst

Category

Object code control

Pragma equivalent

#pragma options [no]roconst

Purpose

Specifies the storage location for constant values.

When **roconst** is in effect, constants are placed in read-only storage. When **noroconst** is in effect, constants are placed in read/write storage.

Syntax

```
           ┌roconst──┐
►►── -q────└noroconst┘──────────────────────────────────────────────────►◄
```

Defaults

- **-qroconst** for all compiler invocations except **cc** and its derivatives. **-qnoroconst** for the **cc** invocation and its derivatives.

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 267
- "-qroptr" on page 270

-qroptr

Category

Object code control

Pragma equivalent

None.

Purpose

Specifies the storage location for constant pointers.

When **-qroptr** is in effect, constant pointers are placed in read-only storage. When **-qnoroptr** is in effect, pointers are placed are placed in read/write storage.

Syntax

```
            ┌─noroptr─┐
►►── -q─────┴─roptr───┴───────────────────────────────────────────────►◄
```

Defaults

-qnoroptr

Usage

A constant pointer is equivalent to an address constant. For example:

```
int* const p = &n;
```

When **-qnoroptr** is in effect, you can change the values of constant pointers without generating errors.

The **-qroptr** can improve runtime performance, save storage, and provide shared access, but code that attempts to modify a read-only constant value generates a memory error. For example, assume the following code, which attempts to change the address that `c1_ptr` points to:

```
char c1 = 10;
char c2 = 20;
char* const c1_ptr = &c1;

int main() {
    *(char**)&c1_ptr = &c2;
}
```

Compiling this code with the **-qroptr** option specified will result in a segmentation fault at run time.

You should not use **-qroptr** for compiled code that will become part of a shared library.

Predefined macros

None.

Related information

- “-qro” on page 267
- “-qroconst” on page 268

-s

Category

Object code control

Pragma equivalent

None.

Purpose

Strips the symbol table, line number information, and relocation information from the output file.

This command is equivalent to the operating system **strip** command.

Syntax

```
►►— -s————————————————————————————————————————————————————————————►◄
```

Defaults

The symbol table, line number information, and relocation information are included in the output file.

Usage

Specifying **-s** saves space, but limits the usefulness of traditional debug programs when you are generating debugging information using options such as **-g**.

Predefined macros

None.

Related information

- “-g” on page 160

-S

Category

Output control

Pragma equivalent

None.

Purpose

Generates an assembler language file for each source file.

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

Syntax

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

Defaults

Not applicable.

Usage

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

```
xlc myprogram.s
```

will invoke the assembler, and if successful, the linker to create an executable file, a.out.

If you specify **-S** with **-E** or **-P**, **-E** or **-P** takes precedence. Order of precedence holds regardless of the order in which they were specified on the command line.

You can use the **-o** option to specify the name of the file produced only if no more than one source file is supplied. For example, the following is *not* valid:

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

Predefined macros

None.

Examples

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

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

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

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

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

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

Related information

- "-E" on page 136
- "-P" on page 244

-qsaveopt

Category

Object code control

Pragma equivalent

None.

Purpose

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

Syntax

```
          ┌─nosaveopt─┐
►►── -q───┴─saveopt───┴────────────────────────────────────────────►◄
```

Defaults

-qnosaveopt

Usage

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

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

```
►►──@(#)──opt──┬─f─┬──invocation──options───────────────────────►◄
               ├─c─┤
               └─C─┘
```

```
►►──@(#)──cfg────config_file_options_list────────────────────────►◄
```

```
►►──@(#)──env────env_var_definition──────────────────────────────►◄
```

where:

f Signifies a Fortran language compilation.

c Signifies a C language compilation.

C Signifies a C++ language compilation.

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

options The list of command line options specified on the command line, with individual options separated by space.

config_file_options_list
The list of options specified by the **options** attribute in all configuration files that take effect in the compilation, separated by space.

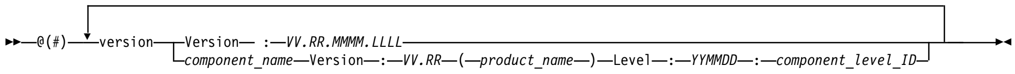
env_var_definition
The environment variables that are used by the compiler. Currently only `XLC_USR_CONFIG` is listed.

Note: You can always use this option, but the corresponding information is only generated when the environment variable `XLC_USR_CONFIG` is set.

For more information about the environment variable **XLC_USR_CONFIG**, see Compile-time and link-time environment variables.

Note: The string of the command-line options is truncated after 64,000 bytes.

Compiler version and release information, as well as the version and level of each component invoked during compilation, are also saved to the object file in the format:



```
         ┌─────────────────────────────────────────────────────────────────────────────────────┐
         ▼                                                                                     │
►►─@(#)──version──┬Version─ :─VV.RR.MMMM.LLLL───────────────────────────────────────────────┬──┴──►◄
                  └component_name─Version─:─VV.RR─(─product_name─)─Level─:─YYMMDD─:─component_level_ID─┘
```

where:

V Represents the version.

R Represents the release.

M Represents the modification.

L Represents the level.

component_name
Specifies the components that were invoked for this compilation, such as the low-level optimizer.

product_name
Indicates the product to which the component belongs (for example, `C/C++` or `Fortran`).

YYMMDD
Represents the year, month, and date of the installed update (PTF). 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 **-qversion** option.

Predefined macros

None.

Examples

Compile `t.c` with the following command:

```
xlc t.c -c -qsaveopt -qhot
```

Issuing the **what** command on the resulting `t.o` object file produces information similar to the following:

```
opt c /opt/IBM/xlc/13.1.3/bin/xlc t.c -c -qsaveopt -qhot
cfg -qlanglvl=extc99 -qcpluscmt -qkeyword=inline -qalias=ansi -D_AIX -D_AIX32
 -D_AIX41 -D_AIX43 -D_AIX50 -D_AIX51 -D_AIX52 -D_AIX53 -D_IBMR2 -D_POWER
version IBM XL C for AIX, V13.1.3
version Version: 13.01.0003.0000
version Driver Version: 13.1.3(C) Level: YYMMDD
version Front End  Version: 13.1.3(C)  Level: YYMMDD
version C Front End Version : 13.1.3(C)  Level: YYMMDD
version High-Level Optimizer Version: 13.1.3(C) and 15.1.3(Fortran)  Level: YYMMDD
version Low-Level Optimizer  Version: 13.1.3(C) and 15.1.3(Fortran) Level: YYMMDD
```

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 (PTFs) you have installed on your system.

Related information

- "-qversion" on page 321

-qshowinc

Category

Listings, messages, and compiler information

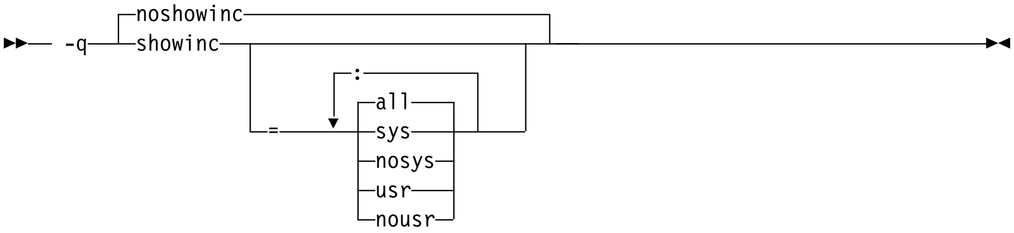
Pragma equivalent

#pragma options [no]showinc

Purpose

When used with **-qsource** option to generate a listing file, selectively shows user or system header files in the source section of the listing file.

Syntax



Defaults

-qnoshowinc: Header files included in source files are not shown in the source listing.

Parameters

all
Shows both user and system include files in the program source listing.

sys
Shows system include files (that is, files included with the `#include <filename>` preprocessor directive) in the program source listing.

usr
Shows user include files (that is, files included with the `#include "filename"` preprocessor directive or with **-qinclude**) in the program source listing.

Specifying **showinc** with no suboptions is equivalent to **-qshowinc=sys : usr** and **-qshowinc=all**. Specifying **noshowinc** is equivalent to **-qshowinc=nosys : nousr**.

Usage

This option has effect only when the **-qlist** or **-qsource** compiler options is in effect.

Predefined macros

None.

Examples

To compile `myprogram.c` so that all included files appear in the source listing, enter:

```
xlc myprogram.c -qsource -qshowinc
```

Related information

- "-qsource" on page 286

-qshowmacros

Category

"Output control" on page 75

Pragma equivalent

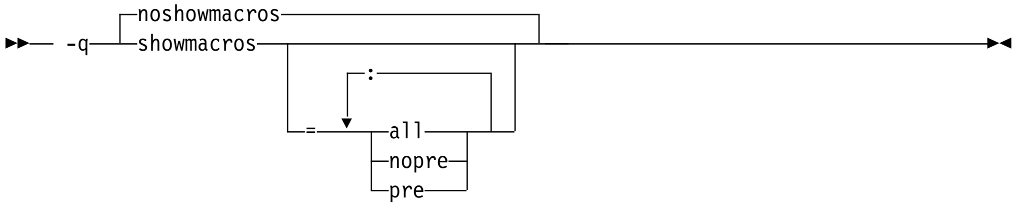
None

Purpose

Emits macro definitions to preprocessed output.

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

Syntax



Defaults

-qnoshowmacros

Parameters

`all`
: Emits all macro definitions to preprocessed output. This is the same as specifying **-qshowmacros**.

pre | nopre
pre emits only predefined macro definitions to preprocessed output. **nopre** suppresses appending these definitions.

Usage

Note the following when using this option:

- This option has no effect unless preprocessed output is generated; for example, by using the **-E** or **-P** options.
- If a macro is defined and subsequently undefined before compilation ends, this macro will not be included in the preprocessed output.
- Only macros defined internally by the preprocessor are considered predefined; all other macros are considered as user-defined.

Related information

- “-E” on page 136
- “-P” on page 244

-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

```
          ┌─showpdf───┐
►►── -q───┴─noshowpdf─┴──────────────────────────────────────────────►◄
```

Defaults

-qshowpdf

Usage

After you run your application with typical data, the profiling information is recorded into a profile-directed feedback (PDF) file (by default, 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 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 247
- "Optimizing your applications" in the *XL C Optimization and Programming Guide*

-qsimd

Category

Optimization and tuning

Pragma equivalent

```
#pragma nosimd
```

Purpose

Controls whether the compiler can automatically take advantage of vector instructions for processors that support them.

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

Syntax

```
                   ┌─auto───┐
►►──-q──simd──=────┴─noauto─┴──────────────────────────────────────────►◄
```

Defaults

Whether **-qsimd** is specified or not, **-qsimd=auto** is implied when both of the following conditions are satisfied; otherwise, **-qsimd=noauto** is implied.

- The optimization level is **-O3** or higher.
- **-qarch** is set to **pwr7** or higher.

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 294.

The **-qsimd=auto** option controls the autosimdization, which was performed by the deprecated **-qhot=simd** option. If you specify **-qhot=simd**, the compiler ignores it and does not issue any warning message.

Specifying the deprecated **-qenablevmx** option has the same effect as specifying **-qsimd=auto**. The compiler does not issue any warning for this.

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. It also sets an appropriate value for **-qarch** to match the architecture that is specified at the compile time. 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 362
- "-qarch" on page 102
- "-qreport" on page 263
- "-qstrict" on page 294

- *Using interprocedural analysis* in the *XL C Optimization and Programming Guide*.

-qskipsrc

Category

"Listings, messages, and compiler information" on page 84

Pragma equivalent

None.

Purpose

When a listing file is generated using the **-qsource** option, **-qskipsrc** can be used to determine whether the source statements skipped by the compiler are shown in the source section of the listing file. Alternatively, the **-qskipsrc=hide** option is used to hide the source statements skipped by the compiler.

Syntax

```
                          ┌show┐
►►── -q──skipsrc──=──┴hide┴──────────────────────────────────►◄
```

Defaults

- -qskipsrc=show

Parameters

show | **hide**

When **show** is in effect, the compiler will display all source statements in the listing. This will result in both true and false paths of the preprocessing directives to be shown.

On the contrary, when **hide** is enabled, all source statements that the compiler skipped will be omitted.

Usage

In general, the **-qskipsrc** option does not control whether the source section is included in the listing file, it only does so when the **-qsource** option is in effect.

To display all source statements in the listing (default option):

```
xlc myprogram.c -qsource -qskipsrc=show
```

To omit source statements skipped by the compiler:

```
xlc myprogram.c -qsource -qskipsrc=hide
```

Predefined macros

None.

Related information

- "-qsource" on page 286
- "-qshowinc" on page 275
- "-qsrcmsg" on page 290

-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

```
          ┌─nosmallstack─┐
►►── -q───┴─smallstack───┴──────────────────────────────────────────────►◄
```

Defaults

-qnosmallstack

Usage

AIX limits the stack size to 256 MB. 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 160
- "-qipa" on page 193
- "-O, -qoptimize" on page 236

-qsmp

Category

Optimization and tuning

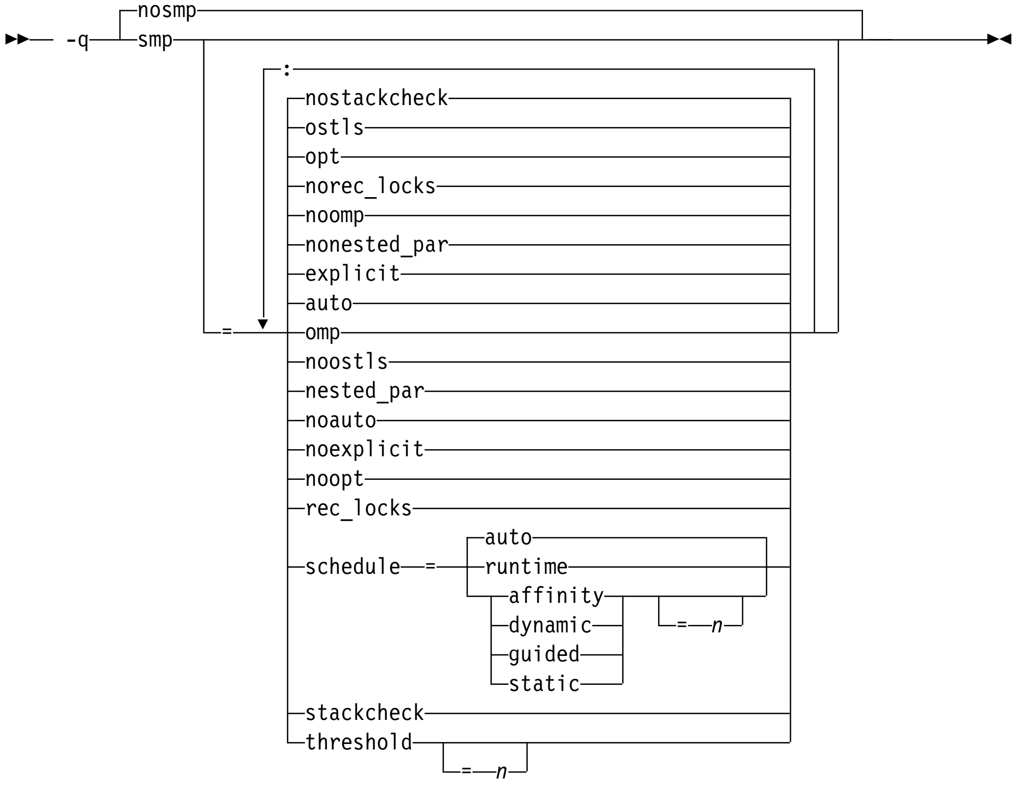
Pragma equivalent

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 SMP or OpenMP directives is optimized. **noauto** is implied if you specify **-qsmp=omp** or **-qsmp=noopt**.

explicit | noexplicit
Enables or disables directives controlling explicit parallelization of loops.

nested_par | nonested_par
By default, the compiler serializes a nested parallel construct. When **nested_par** is in effect, the compiler parallelizes prescriptive nested parallel constructs. This includes not only the loop constructs that are nested within a scoping unit but also parallel constructs in subprograms that are referenced (directly or indirectly) from within other parallel constructs. Note that this suboption has

no effect on loops that are automatically parallelized. In this case, at most one loop in a loop nest (in a scoping unit) will be parallelized.

The setting of the omp_set_nested function or of the OMP_NESTED environment variable overrides the setting of the **-qsmp = nested_par | nonested_par** option.

This suboption should be used with caution. Depending on the number of threads available and the amount of work in an outer loop, inner loops could be executed sequentially even if this option is in effect. Parallelization overhead may not necessarily be offset by program performance gains.

Note: The **-qsmp=nested_par | nonested_par** option has been deprecated and might be removed in a future release. Use the OMP_NESTED environment variable or the omp_set_nested function instead.

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.

ostls| noostls

Enables thread-local storage (TLS) provided by the operating system to be used for **threadprivate** data. You can use the **noostls** suboption to enable non-TLS for **threadprivate**. The **noostls** suboption is provided for compatibility with earlier versions of the compiler.

Note: If you use this suboption, your operating system must support TLS to implement OpenMP **threadprivate** data. Use **noostls** to disable OS level TLS if your operating system does not support it.

rec_locks | norec_locks

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

schedule

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

affinity[=*n*]

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

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

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

The **affinity** scheduling type 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 26.

threshold[=*n*]

When **-qsmp=auto** is in effect, controls the amount of automatic loop parallelization that occurs. The value of *n* represents the minimum amount of work required in a loop in order for it to be parallelized. Currently, the calculation of "work" is weighted heavily by the number of iterations in the loop. In general, the higher the value specified for *n*, the fewer loops are parallelized. Specifying a value of 0 instructs the compiler to parallelize all auto-parallelizable loops, whether or not it is profitable to do so. Specifying a value of 100 instructs the compiler to parallelize only those auto-parallelizable loops that it deems profitable. Specifying a value of greater than 100 will result in more loops being serialized.

n Must be a positive integer of 0 or greater.

If you specify **threshold** with no suboption, the program uses a default value of 100.

Specifying **-qsmp** without suboptions is equivalent to:

```
-qsmp=auto:explicit:opt:noomp:norec_locks:nonested_par:schedule=auto:
nostackcheck:threshold=100:ostls
```

Usage

- Specifying the **omp** suboption always implies **noauto**. Specify **-qsmp=omp:auto** to apply automatic parallelization on OpenMP-compliant applications, as well.
- You should only use **-qsmp** with the **_r**-suffixed invocation commands, to automatically link in all of the threadsafe components. You can use the **-qsmp** option with the non-**_r**-suffixed invocation commands, but you are responsible for linking in the appropriate components. If you use the **-qsmp** option to compile any source file in a program, then you must specify the **-qsmp** option at link time as well, unless you link by using the **ld** command.
- Object files generated with the **-qsmp=opt** option can be linked with object files generated with **-qsmp=noopt**. The visibility within the debugger of the variables in each object file will not be affected by linking.
- The **-qnosmp** default option setting specifies that no code should be generated for parallelization directives, though syntax checking will still be performed. Use **-qignprag=omp:ibm** 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**.

Predefined macros

When **-qsmp** is in effect, _IBMSMP is predefined to a value of 1, which indicates that IBM SMP directives are recognized; otherwise, it is not defined.

Related information

- "-O, -qoptimize" on page 236
- "-qthreaded" on page 306
- "Environment variables for parallel processing" on page 25
- "Pragma directives for parallel processing" on page 380
- "Built-in functions for parallel processing" on page 612

-qsource

Category

Listings, messages, and compiler information

Pragma equivalent

#pragma options [no]source

Purpose

Produces a compiler listing file that includes the source section of the listing and provides additional source information when printing error messages.

Syntax

```
          ┌─nosource─┐
►►── -q───┴─source───┴──────────────────────────────────────────►◄
```

Defaults

-qnosource

Usage

When **-qsource** or **#pragma options source** is in effect, a listing file with the .lst suffix is generated for each source file specified on the command line. For details about the contents of the listing file, see "Compiler listings" on page 19.

You can selectively print parts of the source by using pairs of **#pragma options source** and **#pragma options nosource** preprocessor directives throughout your source program. The source following **#pragma options source** and preceding **#pragma options nosource** is printed.

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

Predefined macros

None.

Examples

The `myprogram.c` file contains the following code:

```
#include <stdio.h>
int main()
{
  printf("Hello World");
}
```

To compile the `myprogram.c` file to produce a compiler listing that includes the source code, enter:

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

The `myprogram.lst` file contains a source section with the code in the `myprogram.c` file:

```
>>>>> SOURCE SECTION <<<<<

1 | # include <stdio.h>
2 |
3 | int main ()
4 | {
5 |   printf("Hello World");
6 | }
```

Related information

- "-qlist" on page 217
- "-qlistopt" on page 221
- "-qprint" on page 259

-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. The **-qsourcetype** option instructs the compiler to not rely on the file name suffix, and to instead assume a source type as specified by the option.

Syntax

```
                        ┌─default────────────┐
►►── -q─sourcetype──=───┼─assembler──────────┼──────────────────────────►◄
                        ├─assembler-with-cpp─┤
                        └─c──────────────────┘
```

Defaults

-qsourcetype=default

Parameters

assembler
All source files following the option are compiled as if they are assembler language source files.

assembler-with-cpp
All source files following the option are compiled as if they are assembler language source files that need preprocessing.

c All source files following the option are compiled as if they are C language source files.

default
The programming language of a source file is implied by its file name suffix.

Usage

If you do not use this option, files must have a suffix of .c to be compiled as C files.

This option applies whether the file system is case-sensitive or not. That is, even in a case-insensitive file system, where `file.c` and `file.C` refer to the same physical file, the compiler still recognizes the case difference of the file name argument on the command line and determines the source type accordingly.

Note that the option only affects files that are specified on the command line *following* the option, but not those that precede the option. Therefore, in the following example:

```
xlc goodbye.C -qsourcetype=c hello.C
```

`hello.C` is compiled as a C source file, but `goodbye.C` is compiled as a C++ file, assuming a C++ compiler is available.

Predefined macros

None.

Examples

To treat the source file `hello.C` as being a C language source file, enter:

```
xlc -qsourcetype=c hello.C
```

-qspeculateabsolutes

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Works with the **-qtocmerge -bl:file** for non-IPA links and with the **-bl:file** for IPA links to disable speculation at absolute addresses.

The **bl:file** is necessary for the compiler to know which addresses are absolutes.

Syntax

```
          ┌─speculateabsolutes───┐
►►── -q───┴─nospeculateabsolutes─┴──────────────────────────────────►◄
```

Defaults

-qspeculateabsolutes

Predefined macros

None.

Related information

- "-qtocmerge" on page 309

-qspill

Category

Compiler customization

Pragma equivalent

#pragma options [no]spill

Purpose

Specifies the size (in bytes) of the register spill space, the internal program storage areas used by the optimizer for register spills to storage.

Syntax

```
►►── -q──spill──=──size─────────────────────────────────────────────►◄
```

Defaults

-qspill=512

Parameters

size
An integer representing the number of bytes for the register allocation spill area.

Usage

If your program is very complex, or if there are too many computations to hold in registers at one time and your program needs temporary storage, you might need to increase this area. Do not enlarge the spill area unless the compiler issues a message requesting a larger spill area. In case of a conflict, the largest spill area specified is used.

Predefined macros

None.

Examples

If you received a warning message when compiling `myprogram.c` and want to compile it specifying a spill area of 900 entries, enter:

```
xlc myprogram.c -qspill=900
```

-qsrcmsg

Category

Listings, messages, and compiler information

Pragma equivalent

#pragma options [no]srcmsg

Purpose

Adds the corresponding source code lines to diagnostic messages generated by the compiler.

When **nosrcmsg** is in effect, the error message simply shows the file, line and column where the error occurred. When **srcmsg** is in effect, the compiler reconstructs the source line or partial source line to which the diagnostic message refers and displays it before the diagnostic message. A pointer to the column position of the error may also be displayed.

Syntax

```
           ┌─nosrcmsg─┐
►►── -q────┴─srcmsg───┴──────────────────────────────────────────────►◄
```

Defaults

-qnosrcmsg

Usage

When **srcmsg** is in effect, the reconstructed source line represents the line as it appears after macro expansion. At times, the line may be only partially reconstructed. The characters `"...."` at the start or end of the displayed line indicate that some of the source line has not been displayed.

Use **-qnosrcmsg** to display concise messages that can be parsed.

Predefined macros

None.

-qstackprotect

Category

"Object code control" on page 79

Pragma equivalent

None.

Purpose

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

Syntax

```
          ┌─nostackprotect──────────────────────┐
►►── -q───┴─stackprotect────=──────┬─all──────┬──┴──────────────────────────────────►◄
                                   └─size──=──N─┘
```

Defaults

-qnostackprotect

Parameters

`all`
: Protects all functions whether or not functions have vulnerable objects. This option is not set by default.

`size=`*`N`*
: Protects all functions containing automatic objects with size greater than or equal to *N* bytes. The default size is 8 byteswhen **-qstackprotect** is enabled.

Usage

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

To generate code to protect all functions with vulnerable objects, enter the following command:

```
xlc myprogram.c -qstackprotect=all
```

To generate code to protect functions with objects of certain size, enter the following command with the **`size=`** parameter set to the object size indicated in bytes:

```
xlc myprogram.c -qstackprotect=size=8
```

Notes:

- Because of the dependency on **libc.a** in AIX, this option requires AIX 6.1/TL4 or higher.
- If the link step fails with a message that indicates **__ssp_canary_word** is undefined, you have probably used an unsupported level of AIX.

Predefined macros

None.

Related information

- "-qinfo" on page 178

-qstatsym

Category

Object code control

Pragma equivalent

None.

Purpose

Adds user-defined, nonexternal names that have a persistent storage class, such as initialized and uninitialized static variables, to the symbol table of the object file.

Syntax

```
          ┌─nostatsym─┐
►►─ -q────┴─statsym───┴────────────────────────────────────────────►◄
```

Defaults

-qnostatsym

Usage

When **-qnostatsym** is specified, static variables are not added to the symbol table. However, static functions are added to the symbol table.

Predefined macros

None.

-qstdinc

Category

Input control

Pragma equivalent

#pragma options [no]stdinc

Purpose

Specifies whether the standard include directories are included in the search paths for system and user header files.

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

- The directory specified in the configuration file for the XL C header files (this is normally /opt/IBM/xlc/13.1.3/include/) or by the **-qc_stdinc** option
- The directory specified in the configuration file for the system header files (this is normally /usr/include/), or by the **-qc_stdinc** option

When **-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 **-qinclude** option

Syntax

```
           ┌─stdinc───┐
▶▶── -q────┴─nostdinc─┴──────────────────────────────────────────▶◀
```

Defaults

-qstdinc

Usage

The search order of header files is described in "Directory search sequence for included files" on page 12.

This option only affects search paths for header files included with a relative name; if a full (absolute) path name is specified, this option has no effect on that path name.

The last valid pragma directive remains in effect until replaced by a subsequent pragma.

Predefined macros

None.

Examples

To compile `myprogram.c` so that *only* the directory `/tmp/myfiles` (in addition to the directory containing `myprogram.c`) is searched for the file included with the `#include "myinc.h"` directive, enter:

```
xlc myprogram.c -qnostdinc -I/tmp/myfiles
```

Related information

- "-qc_stdinc" on page 125
- "-I" on page 172

- “Directory search sequence for included files” on page 12

-qstrict

Category

Optimization and tuning

Pragma equivalent

#pragma options [no]strict

#pragma option_override (*function_name*, "opt (*suboption_list*)")

Purpose

Ensures that optimizations that are done by default at the **-O3** and higher optimization levels, and, optionally at **-O2**, do not alter the semantics of a program.

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

Syntax

```
►►── -q──┬─nostrict──────────────────────────────────────┬──────────────►◄
         └─strict──┬─────────────────────────────────────┘
                   │      ┌─ :──────────────────────────┐ │
                   │      ▼                             │ │
                   └─=────┬─all───────────────────┬─────┴─┘
                          ├─none──────────────────┤
                          ├─precision─────────────┤
                          ├─noprecision───────────┤
                          ├─exceptions────────────┤
                          ├─noexceptions──────────┤
                          ├─ieeefp────────────────┤
                          ├─noieeefp──────────────┤
                          ├─nans──────────────────┤
                          ├─nonans────────────────┤
                          ├─infinities────────────┤
                          ├─noinfinities──────────┤
                          ├─subnormals────────────┤
                          ├─nosubnormals──────────┤
                          ├─zerosigns─────────────┤
                          ├─nozerosigns───────────┤
                          ├─operationprecision────┤
                          ├─nooperationprecision──┤
                          ├─vectorprecision───────┤
                          ├─novectorprecision─────┤
                          ├─order─────────────────┤
                          ├─noorder───────────────┤
                          ├─association───────────┤
                          ├─noassociation─────────┤
                          ├─reductionorder────────┤
                          ├─noreductionorder──────┤
                          ├─guards────────────────┤
                          ├─noguards──────────────┤
                          ├─library───────────────┤
                          └─nolibrary─────────────┘
```

Defaults

- **-qstrict** or **-qstrict=all** is always in effect when the **-qnoopt** or **-O0** optimization level is in effect
- **-qstrict** or **-qstrict=all** is the default when the **-O2** or **-O** optimization level is in effect
- **-qnostrict** or **-qstrict=none** is the default when the **-O3** or higher optimization level is in effect

Parameters

The **-qstrict** suboptions include the following:

all | none

all disables all semantics-changing transformations, including those controlled by the **ieeefp**, **order**, **library**, **precision**, and **exceptions** suboptions. **none** enables these transformations.

precision | noprecision

precision disables all transformations that are likely to affect floating-point precision, including those controlled by the **subnormals**, **operationprecision**,

vectorprecision, **association**, **reductionorder**, and **library** suboptions. **noprecision** enables these transformations.

`exceptions | noexceptions`
exceptions disables all transformations likely to affect exceptions or be affected by them, including those controlled by the **nans**, **infinities**, **subnormals**, **guards**, and **library** suboptions. **noexceptions** enables these transformations.

`ieeefp | noieeefp`
ieeefp disables transformations that affect IEEE floating-point compliance, including those controlled by the **nans**, **infinities**, **subnormals**, **zerosigns**, **vectorprecision**, and **operationprecision** suboptions. **noieeefp** enables these transformations.

`nans | nonans`
nans disables transformations that may produce incorrect results in the presence of, or that may incorrectly produce IEEE floating-point NaN (not-a-number) values. **nonans** enables these transformations.

`infinities | noinfinities`
infinities disables transformations that may produce incorrect results in the presence of, or that may incorrectly produce floating-point infinities. **noinfinities** enables these transformations.

`subnormals | nosubnormals`
subnormals disables transformations that may produce incorrect results in the presence of, or that may incorrectly produce IEEE floating-point subnormals (formerly known as denorms). **nosubnormals** enables these transformations.

`zerosigns | nozerosigns`
zerosigns disables transformations that may affect or be affected by whether the sign of a floating-point zero is correct. **nozerosigns** enables these transformations.

`operationprecision | nooperationprecision`
operationprecision disables transformations that produce approximate results for individual floating-point operations. **nooperationprecision** enables these transformations.

`vectorprecision | novectorprecision`
vectorprecision disables vectorization in loops where it might produce different results in vectorized iterations than in nonvectorized residue iterations. **vectorprecision** ensures that every loop iteration of identical floating-point operations on identical data produces identical results.

novectorprecision enables vectorization even when different iterations might produce different results from the same inputs.

`order | noorder`
order disables all code reordering between multiple operations that may affect results or exceptions, including those controlled by the **association**, **reductionorder**, and **guards** suboptions. **noorder** enables code reordering.

`association | noassociation`
association disables reordering operations within an expression. **noassociation** enables reordering operations.

`reductionorder | noreductionorder`
reductionorder disables parallelizing floating-point reductions. **noreductionorder** enables parallelizing these reductions.

guards | noguards

Specifying **-qstrict=guards** has the following effects:

- The compiler does not move operations past guards, which control whether the operations are executed. That is, the compiler does not move operations past guards of the if statements, out of loops, or past guards of function calls that might end the program or throw an exception.
- When the compiler encounters if expressions that contain pointer wraparound checks that can be resolved at compile time, the compiler does not remove the checks or the enclosed operations. The pointer wraparound check compares two pointers that have the same base but have constant offsets applied to them.

Specifying **-qstrict=noguards** has the following effects:

- The compiler moves operations past guards.
- The compiler evaluates if expressions according to language standards, in which pointer wraparounds are undefined. The compiler removes the enclosed operations of the if statements when the evaluation results of the if expressions are false.

library | nolibrary

library disables transformations that affect floating-point library functions; for example, transformations that replace floating-point library functions with other library functions or with constants. **nolibrary** enables these transformations.

Usage

The **all**, **precision**, **exceptions**, **ieeefp**, and **order** suboptions and their negative forms are group suboptions that affect multiple, individual suboptions. For many situations, the group suboptions will give sufficient granular control over transformations. Group suboptions act as if either the positive or the no form of every suboption of the group is specified. Where necessary, individual suboptions within a group (like **subnormals** or **operationprecision** within the **precision** group) provide control of specific transformations within that group.

With **-qnostrict** or **-qstrict=none** in effect, the following optimizations are turned on:

- Code that may cause an exception may be rearranged. The corresponding exception might happen at a different point in execution or might not occur at all. (The compiler still tries to minimize such situations.)
- Floating-point operations may not preserve the sign of a zero value. (To make certain that this sign is preserved, you also need to specify **-qfloat=rrm**, **-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=fltint:rsqrt**. You can turn off the optimization functions by using the **-qstrict** option or **-qfloat=nofltint: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=nofltint:norsqrt:rngchk**. **-qnostrict** or **-qstrict=none** sets **-qfloat=fltint:rsqrt:norngchk**.

- **-qstrict=operationprecision** or **-qstrict=exceptions** sets **-qfloat=nofltint**. Specifying both **-qstrict=nooperationprecision** and **-qstrict=noexceptions** sets **-qfloat=fltint**.
- **-qstrict=infinities**, **-qstrict=operationprecision**, or **-qstrict=exceptions** sets **-qfloat=norsqrt**.
- **-qstrict=noinfinities:nooperationprecision:noexceptions** sets **-qfloat=rsqrt**.
- **-qstrict=nans**, **-qstrict=infinities**, **-qstrict=zerosigns**, or **-qstrict=exceptions** sets **-qfloat=rngchk**. Specifying all of **-qstrict=nonans:nozerosigns:noexceptions** or **-qstrict=noinfinities:nozerosigns:noexceptions**, or any group suboptions that imply all of them, sets **-qfloat=norngchk**.

Note: For details about the relationship between **-qstrict** suboptions and their **-qfloat** counterparts, see "-qfloat" on page 146.

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, range checking is turned off (**-qfloat=fltint**), 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=fltint: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 278
- "-qfloat" on page 146

- “-qhot” on page 169
- “-O, -qoptimize” on page 236

-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

```
             ┌─strict_induction───┐
►►── -q──────┴─nostrict_induction─┴──────────────────────────────────────►◄
```

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 236

-qsuppress

Category

Listings, messages, and compiler information

Pragma equivalent

None.

Purpose

Prevents specific informational or warning messages from being displayed or added to the listing file, if one is generated.

Syntax

```
          ┌─nosuppress──────────────────────────┐
          │             ┌─:──────────────┐      │
          │             ▼                │      │
►►── -q ──┴─suppress──=───message_identifier─┴──┴──────────────────────────────►◄
```

Defaults

-qnosuppress: All informational and warning messages are reported, unless set otherwise with the **-qflag** option.

Parameters

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

`15`*`dd-number`*

where:

15 Is the compiler product identifier.

dd Is the two-digit code representing the compiler component that produces the message. See "Compiler message format" on page 17 for descriptions of these codes.

number
Is the message number.

Usage

You can only suppress information (I) and warning (W) messages. You cannot suppress other types of messages, such as (S) and (U) level messages. Note that informational and warning messages that supply additional information to a severe error cannot be disabled by this option.

To suppress all informational and warning messages, you can use the **-w** option.

To suppress IPA messages, enter **-qsuppress** before **-qipa** on the command line.

The **-qhaltonmsg** compiler option has precedence over **-qsuppress**. If both **-qhaltonmsg** and **-qsuppress** are specified, messages that are suppressed by **-qsuppress** are also printed.

The **-qnosuppress** compiler option cancels previous settings of **-qsuppress**.

Predefined macros

None.

Examples

If your program normally results in the following output:

```
"myprogram.c", line 1.1:1506-224 (I) Incorrect #pragma ignored
```

you can suppress the message by compiling with:

```
xlc myprogram.c -qsuppress=1506-224
```

Related information

- "-qflag" on page 145
- "-qhaltonmsg" on page 166

-qsymtab

Category

Error checking and debugging

Pragma equivalent

None.

Purpose

Determines the information that appears in the symbol table.

Syntax

```
►►── -q──symtab──=──┬─unref──┬──────────────────────────────────────────────►◄
                    └─static─┘
```

Defaults

Static variables and unreferenced `typedef`, structure, union, and enumeration declarations are not included in the symbol table of the object file.

Parameters

unref
When used with the **-g** option, specifies that debugging information is included for unreferenced `typedef` declarations, `struct`, `union`, and `enum` type definitions in the symbol table of the object file. This suboption is equivalent to **-qdbxextra**.

Using **-qsymtab=unref** may make your object and executable files larger.

static
Adds user-defined, nonexternal names that have a persistent storage class, such as initialized and uninitialized static variables, to the symbol table of the object file. This suboption is equivalent to **-qstatsym**.

Predefined macros

None.

Examples

To compile `myprogram.c` so that static symbols are added to the symbol table, enter:

```
xlc myprogram.c -qsymtab=static
```

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

```
xlc myprogram.c -g -qsymtab=unref
```

Related information

- “-g” on page 160
- “-qdbxextra” on page 130
- “-qstatsym” on page 292

-qsyntaxonly

Category

Error checking and debugging

Pragma equivalent

None.

Purpose

Performs syntax checking without generating an object file.

Syntax

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

Defaults

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

Usage

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

The **-qsyntaxonly** option suppresses only the generation of an object file. All other files, such as listing files, are still produced if their corresponding options are set.

Predefined macros

None.

Examples

To check the syntax of `myprogram.c` without generating an object file, enter:

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

Related information

- “-C, -C!” on page 115
- “-c” on page 114
- “-E” on page 136
- “-o” on page 235
- “-P” on page 244

-t

Category

Compiler customization

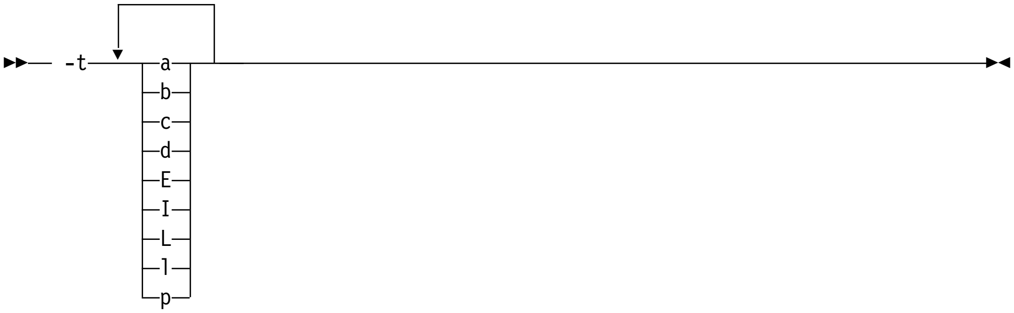
Pragma equivalent

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	The compiler front end	xlcentry
d	The disassembler	dis
E	The CreateExportList utility	CreateExportList
I (uppercase i)	The high-level optimizer, compile step	ipa

Parameter	Description	Component name
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 110

-qtabsize

Category

Language element control

Pragma equivalent

#pragma options tabsize

Purpose

Sets the default tab length, for the purposes of reporting the column number in error messages.

Syntax

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

Defaults

-qtabsize=8

Parameters

number
The number of character spaces representing a tab in your source program.

Usage

This option only affects error messages that specify the column number at which an error occurred.

Predefined macros

None.

Examples

To compile `myprogram.c` so the compiler considers tabs as having a width of one character, enter:

```
xlc myprogram.c -qtabsize=1
```

In this case, you can consider one character position (where each character and each tab equals one position, regardless of tab length) as being equivalent to one character column.

-qtbtable

Category

Object code control

Pragma equivalent

#pragma options tbtable

Purpose

Controls the amount of debugging traceback information that is included in the object files.

Many performance measurement tools require a full traceback table to properly analyze optimized code. If a traceback table is generated, it is placed in the text segment at the end of the object code, and contains information about each function, including the type of function, as well as stack frame and register information.

Syntax

```
                         ┌─full──┐
►►── -q──tbtable──=──────┼─none──┼─────────────────────────────────────────►◄
                         └─small─┘
```

Defaults

- **-qtbtable=full**
- **-qtbtable=small** when **-O** or higher optimization is in effect

Parameters

full
A full traceback table is generated, complete with name and parameter information.

none
No traceback table is generated. The stack frame cannot be unwound so exception handling is disabled.

small
The traceback table generated has no name or parameter information, but otherwise has full traceback capability. This suboption reduces the size of the program code.

Usage

The **#pragma options** directive must be specified before the first statement in the compilation unit.

Predefined macros

None.

Related information

- "-g" on page 160

-qthreaded

Category

Object code control

Pragma equivalent

None.

Purpose

Indicates to the compiler whether it must generate threadsafe code.

Always use this option when compiling or linking multithreaded applications. This option does not make code threadsafe, but it will ensure that code already threadsafe will remain so after compilation and linking. It also ensures that all optimizations are threadsafe.

Syntax

```
          ┌─nothreaded─┐
►►── -q───┴─threaded───┴────────────────────────────────────────►◄
```

Defaults

- **-qnothreaded** for all invocation commands except those with the **_r** suffix
- **-qthreaded** for all **_r**-suffixed invocation commands

Usage

This option applies to both compile and linker operations.

To maintain thread safety, a file compiled with the **-qthreaded** option, whether explicitly by option selection or implicitly by choice of **_r** compiler invocation mode, must also be linked with the **-qthreaded** option.

Predefined macros

None.

Related information

- "-qsmp" on page 281

-qtimestamps

Category

"Output control" on page 75

Pragma equivalent

None.

Purpose

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

Syntax

```
           ┌─timestamps───┐
►►── -q────┴─notimestamps─┴──────────────────────────────────────────►◄
```

Defaults

-qtimestamps

Usage

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

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

-qtls

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

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

Defaults

-qtls=unsupported

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

Parameters

`unsupported`
The `__thread` keyword is not recognized and thread-local storage is not enabled. This suboption is equivalent to **-qnotls**.

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

`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

- "-qpic" on page 254
- "The __thread storage class specifier" in the *XL C Language Reference*

-qtocmerge

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Enables TOC merging to reduce TOC pointer loads and improves the scheduling of external loads.

Syntax

```
           ┌─notocmerge─┐
►►── -q────┴─tocmerge───┴──────────────────────────────────────────────►◄
```

Defaults

-qnotocmerge

Usage

To use **-qtocmerge**, you must also use the **-bImportfile** linker option to specify the name of the file from which the compiler reads.

Predefined macros

None.

-qtrigraph

Category

Language element control

Pragma equivalent

None.

Purpose

Enables the recognition of trigraph key combinations to represent characters not found on some keyboards.

Syntax

```
           ┌─trigraph───┐
►►── -q────┴─notrigraph─┴──────────────────────────────────────────────►◄
```

Defaults

-qtrigraph

Usage

A trigraph is a three-key character combination that let you produce a character that is not available on all keyboards. For details, see "Trigraph sequences" in the *XL C Language Reference*.

Predefined macros

None.

Related information

- "Trigraph sequences" in the *XL C Language Reference*
- “-qdigraph” on page 132
- “-qlanglvl” on page 206

-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

```
            ┌─balanced─┐
►►── -q──tune──=──┼─auto─────┼──┬───────────────────┬─────────────────►◄
            ├─ppc970───┤  │      ┌─st───────┐ │
            ├─pwr4─────┤  └─:──┼─balanced─┼─┘
            ├─pwr5─────┤         ├─smt2─────┤
            ├─pwr6─────┤         ├─smt4─────┤
            ├─pwr7─────┤         └─smt8─────┘
            └─pwr8─────┘
```

Defaults

-qtune=balanced:balanced when no valid **-qarch** setting is in effect. Otherwise, the default depends on the effective **-qarch** setting. For details, see Table 25 on page 312.

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.

balanced
Optimizations are tuned across a selected range of recent hardware.

ppc970
Optimizations are tuned for the PowerPC 970 processor.

pwr4
Optimizations are tuned for the POWER4 hardware platforms.

pwr5
Optimizations are tuned for the POWER5 hardware platforms.

pwr6
Optimizations are tuned for the POWER6 hardware platforms.

pwr7
Optimizations are tuned for the POWER7 or POWER7+ hardware platforms.

pwr8
Optimizations are tuned for the POWER8 hardware platforms.

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.

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

If you want your program to run on more than one architecture, but to be tuned to a particular architecture, you can use a combination of the **-qarch** and **-qtune** options. These options are primarily of benefit for floating-point intensive programs.

By arranging (scheduling) the generated machine instructions to take maximum advantage of hardware features such as cache size and pipelining, **-qtune** can improve performance. It only has an effect when used in combination with options that enable optimization.

A particular **SMT** suboption is valid if the effective **-qarch** option supports the specified **SMT** mode. The acceptable combinations of the **-qarch** and **SMT** tune options are listed in Table 25. The compiler ignores any invalid **-qarch**/**-qtune SMT** combination.

Although changing the **-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.

Acceptable combinations of **-qarch** and **-qtune** are shown in the following table.

Table 25. Acceptable **-qarch/-qtune** *combinations*

-qarch option	Default -qtune setting	Available -qtune CPU settings	Available -qtune SMT settings
ppc	balanced:balanced	auto | pwr4 | pwr5 | pwr6 | pwr7 | pwr8 | ppc970 | balanced	balanced | st
ppcgr	balanced:balanced	auto | pwr4 | pwr5 | pwr6 | pwr7 | pwr8 | ppc970 | balanced	balanced | st
ppc64	balanced:balanced	auto | pwr4 | pwr5 | pwr6 | pwr7 | pwr8 | ppc970 | balanced	balanced | st
ppc64gr	balanced:balanced	auto | pwr4 | pwr5 | pwr6 | pwr7 | pwr8 | ppc970 | balanced	balanced | st
ppc64grsq	balanced:balanced	auto | pwr4 | pwr5 | pwr6 | pwr7 | pwr8 | ppc970 | balanced	balanced | st
ppc64v	balanced:balanced	auto | ppc970 | pwr6 | pwr7 | pwr8 | balanced	balanced | st
ppc970	ppc970:st	auto | ppc970 | balanced	balanced | st
pwr4	pwr4:st	auto | pwr4 | pwr5 | pwr6 | pwr7 | pwr8 | ppc970 | balanced	balanced | st

Table 25. Acceptable **-qarch/-qtune** *combinations (continued)*

-qarch option	Default -qtune setting	Available -qtune CPU settings	Available -qtune SMT settings
pwr5	pwr5:st	auto | pwr5 | pwr6 | pwr7 | pwr8 | balanced	balanced | st
pwr5x	pwr5:st	auto | pwr5 | pwr6 | pwr7 | pwr8 | balanced	balanced | st | smt2
pwr6	pwr6:st	auto | pwr6 | pwr7 | pwr8 | balanced	balanced | st | smt2
pwr6e	pwr6:st	auto | pwr6 | balanced	balanced | st
pwr7	pwr7:st	auto | pwr7 | pwr8 | balanced	balanced | st | smt2 | smt4
pwr8	pwr8:st	auto | pwr8 | balanced	balanced | st | smt2 | smt4 | smt8

Predefined macros

None.

Examples

To specify that the executable program `testing` compiled from `myprogram.c` is to be optimized for a POWER7 hardware platform, enter:

```
xlc -o testing myprogram.c -qtune=pwr7
```

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

- “-qarch” on page 102
- “-q32, -q64” on page 94
- “Specifying compiler options for architecture-specific compilation” on page 9
- "Optimizing your applications" in the *XL C Optimization and Programming Guide*

-U

Category

Language element control

Pragma equivalent

None.

Purpose

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

Syntax

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

Defaults

Many macros are predefined by the compiler; see Chapter 6, "Compiler predefined macros," on page 403 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 126

-qunroll

Category

Optimization and tuning

Pragma equivalent

#pragma options [no]unroll[= yes|no|auto|n]

#pragma unroll

Purpose

Controls loop unrolling, for improved performance.

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

Syntax

Option syntax

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

Pragma syntax

```
►►─#─pragma──┬─nounroll──────────────┬────────────────────────────►◄
             └─unroll──┬───────────┬─┘
                       └─(─n─)─────┘
```

Defaults

-qunroll=auto

Parameters

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

auto (option only)
Instructs the compiler to perform basic loop unrolling.

yes (option only)
Instructs the compiler to search for more opportunities for loop unrolling than that performed with **auto**. In general, this suboption has more chances to increase compile time or program size than **auto** processing, but it might also improve your application's performance.

no (option only)
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**.

-qnounroll is equivalent to **-qunroll=no**.

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 must appear immediately before the loop or the **#pragma block_loop** directive to take effect.

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.

The **#pragma unroll** and **#pragma nounroll** directives can only be used on `for` loops or **#pragma block_loop** directives. They cannot be applied to `do while` and `while` loops.

The loop structure must meet the following conditions:

- There must be only one loop counter variable, one increment point for that variable, and one termination variable. These cannot be altered at any point in the loop nest.
- Loops cannot have multiple entry and exit points. The loop termination must be the only means to exit the loop.
- Dependencies in the loop must not be "backwards-looking". For example, a statement such as `A[i][j] = A[i -1][j + 1] + 4` must not appear within the loop.

Predefined macros

None.

Examples

In the following example, the **#pragma unroll(3)** directive on the first `for` loop requires the compiler to replicate the body of the loop three times. The **#pragma unroll** on the second `for` loop allows the compiler to decide whether to perform unrolling.

```
#pragma unroll(3)
for( i=0;i < n; i++)
{
      a[i] = b[i] * c[i];
}

#pragma unroll
for( j=0;j < n; j++)
{
      a[j] = b[j] * c[j];

}
```

In this example, the first **#pragma unroll(3)** directive results in:

```
i=0;
if (i>n-2) goto remainder;
for (; i<n-2; i+=3) {
  a[i]=b[i] * c[i];
  a[i+1]=b[i+1] * c[i+1];
  a[i+2]=b[i+2] * c[i+2];
}
if (i<n) {
  remainder:
  for (; i<n; i++) {
    a[i]=b[i] * c[i];
  }
}
```

Related information

- "#pragma block_loop" on page 340
- "#pragma loopid" on page 357
- "#pragma stream_unroll" on page 374
- "#pragma unrollandfuse" on page 375

-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

```
            ┌─unwind───┐
►►── -q─────┴─nounwind─┴──────────────────────────────────────────────►◄
```

Defaults

-qunwind

Usage

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

Predefined macros

None.

-qupconv

Category

Portability and migration

Pragma equivalent

#pragma options [no]upconv

Purpose

Specifies whether the `unsigned` specification is preserved when integral promotions are performed.

When **noupconv** is in effect, any `unsigned` type smaller than an `int` is converted to `int` during integral promotions. When **upconv** is in effect, these types are converted to `unsigned int` during integral promotions. The promotion rule does not apply to types that are larger than `int`.

Syntax

```
         ┌─noupconv─┐
►►── -q──┴─upconv───┴──────────────────────────────────────────────►◄
```

Defaults

- **-qnoupconv** for all language levels except **classic** or **extended**
- **-qupconv** when the **classic** or **extended** language levels are in effect

Usage

Sign preservation is provided for compatibility with older dialects of C. The ANSI C standard requires value preservation as opposed to sign preservation.

Predefined macros

None.

Examples

To compile `myprogram.c` so that all `unsigned` types smaller than `int` are converted to `unsigned int`, enter:

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

The following short listing demonstrates the effect of **-qupconv**:

```
#include <stdio.h>
int main(void) {
  unsigned char zero = 0;
  if (-1 <zero)
    printf("Value-preserving rules in effect\n");
  else
    printf("Unsignedness-preserving rules in effect\n");
  return 0;
}
```

Related information

- "Usual arithmetic conversions" in the *XL C Language Reference*
- “-qlanglvl” on page 206

-qutf

Category

Language element control

Pragma equivalent

None.

Purpose

Enables recognition of UTF literal syntax.

Syntax

```
►►— -q——┬noutf┬——————————————————————————————————————————————————————►◄
        └utf——┘
```

Defaults

-qnoutf

Usage

The compiler uses **iconv** library routine to convert the source file to Unicode. If the source file cannot be converted, the compiler will ignore the **-qutf** option and issue a warning.

Predefined macros

None.

Related information

- "UTF literals" in the *XL C Language Reference*

-v, -V

Category

Listings, messages, and compiler information

Pragma equivalent

None.

Purpose

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

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

Syntax

```
►►──┬─ -v─┬──────────────────────────────────────────────────►◄
    └─ -V─┘
```

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 93

-qvecnvol

Category

Portability and migration

Pragma equivalent

None.

Purpose

Specifies whether to use volatile or nonvolatile vector registers.

Syntax

```
          ┌─novecnvol─┐
►►── -q───┴─vecnvol───┴──────────────────────────────────────►◄
```

Defaults

-qnovecnvol

Usage

Volatile vector registers are those whose value is not preserved across function calls or across save context, jump or switch context system library functions. When **-qvecnvol** is in effect, the compiler uses both volatile and nonvolatile vector registers. When **-qnovecnvol** is in effect, the compiler uses only volatile vector registers.

This option is required for programs where there is risk of interaction between modules built with AIX libraries before AIX 5.3TL3 and vector register use. Restricting the compiler to use only volatile registers will make your vector programs safe but it potentially forces the compiler to store vector data to memory more often and therefore results in reducing performance.

Notes:

- This option requires platforms that support vector instructions.
- The **-qnovecnvol** option performs independently from **-qsimd=auto | noauto**, **-qaltivec | -qnoaltivec** and **pragma=nosimd**.
- Before AIX 5.3TL3, by default only 20 volatile registers (vr0-vr19) are used, and 12 nonvolatile vector registers (vr20 - vr31) are not used. You can use these registers only when **-qvecnvol** is in effect.
- **-qvecnvol** should be enabled only when no legacy code that saves and restores nonvolatile registers is involved. Using **-qvecnvol** and linking with legacy code, may result runtime failure.

Predefined macros

None.

Related information

- “-qaltivec” on page 101
- “-qsimd” on page 278

-qversion

Category

Listings, messages, and compiler information

Pragma equivalent

None.

Purpose

Displays the version and release of the compiler being invoked.

Syntax

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

Defaults

-qnoversion

Parameters

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

Usage

When you specify **-qversion**, the compiler displays the version information and exits; compilation is stopped. If you want to save this information to the output object file, you can do so with the **-qsaveopt -c** options.

-qversion specified without the **verbose** suboption shows compiler information in the format:

```
product_nameVersion: VV.RR.MMMM.LLLL
```

where:

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

For more details, see Example 1.

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

```
component_name Version: VV.RR(product_name) Level: component_build_date ID:
component_level_ID
```

where:

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

component_build_date
: Represents the build date of the installed component.

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

For more details, see Example 2.

Predefined macros

None.

Example 1

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

```
IBM XL C/C++ for AIX, V13.1.3 (5765-J06; 5725-C71)

Version: 13.01.0002.0000
```

Example 2

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

```
IBM XL C/C++ for AIX, V13.1.3 V13.1.3 (5765-J06; 5725-C71)
Version: 13.01.0003.0000
Driver Version: 13.1.3(C/C++) Level: 150508
ID: _dRic8vWfEeSjz7qEhQiYJQ
C Front End Version: 13.1.3(C/C++) Level: 150506
ID: _GyiUoOiLEeSbzZ-i2Itj4A
High-Level Optimizer Version: 13.1.3(C/C++) and 15.1.3(Fortran)
Level: 150512 ID: _nAVYcvkLEeSjz7qEhQiYJQ
Low-Level Optimizer Version: 13.1.3(C/C++) and 15.1.3(Fortran)
Level: 150511 ID: _X1GWsPhCEeSjz7qEhQiYJQ
```

Related information

- "-qsaveopt" on page 272

-qvisibility

Category

Optimization and tuning

Pragma equivalent

#pragma GCC visibility push (default | protected | hidden | internal)

#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 **-qvisibility** option if they do not get visibility attributes from pragma directives, explicitly specified attributes, or propagation rules.

Syntax

```
                        ┌─unspecified─┐
►►── -q──visibility──=──┼─default─────┼───────────────────────────────────►◄
                        ├─hidden──────┤
                        ├─protected───┤
                        └─internal────┘
```

Defaults

-qvisibility=unspecified

Parameters

unspecified
Indicates that the affected external linkage entities do not have visibility attributes. Whether these entities are exported in shared libraries depends on the specified export list or the one that is generated by the compiler.

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.

internal
Indicates that the affected external linkage entities have the internal visibility attribute. These entities are not exported in shared libraries, and their addresses are not available to other modules in shared libraries.

Restriction: In this release, the hidden and internal visibility attributes are the same. The addresses of the entities that are specified with either of these visibility attributes can be referenced indirectly through pointers.

Usage

The **-qvisibility** 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.

Note: On the AIX platform, entity preemption occurs only when runtime linking is used. For details, see "Linking a library to an application" in the *XL C Optimization and Programming Guide*. Visibility attributes are supported on AIX 6.1 TL8, AIX 7.1 TL2, AIX 7.2, and higher.

Predefined macros

None.

Examples

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

```
xlc myprogram.c -qvisibility=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

- "-qmkshrobj" on page 233
- "-G" on page 163
- "#pragma GCC visibility push, #pragma GCC visibility pop" on page 349
- "Using visibility attributes (IBM extension)" in the *XL C Optimization and Programming Guide*
- "External linkage", "The visibility variable attribute (IBM extension)", "The visibility function attribute (IBM extension)", in the *XL C Language Reference*

-w

Category

Listings, messages, and compiler information

Pragma equivalent

None.

Purpose

Suppresses warning messages.

This option is equivalent to specifying **-qflag=e : e**.

Syntax

```
►►── -w──────────────────────────────────────────────────────────────►◄
```

Defaults

All informational and warning messages are reported.

Usage

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

Predefined macros

None.

Examples

Consider the file myprogram.c.

```
#include <stdio.h>
int main()
 { char* greeting = "hello world";
   printf("%d \n", greeting);
   return 0;
}
```

- If you compile myprogram.c without the -w option, the compiler issues a warning message.

  ```
  xlC myprogram.c
  ```

  Output:

  ```
  "5:18: warning: format specifies type 'int' but the argument has type 'char *' [-Wformat]
  printf("%d \n", greeting);
  ~~ ^~~~~~
  
  %s
  1 warning generated."
  ```

- If you compile myprogram.c with the -w option, the warning message is suppressed.

  ```
  xlC myprogram.c -w
  ```

Related information

- "-qflag" on page 145
- "-qsuppress" on page 299

-W

Category

Compiler customization

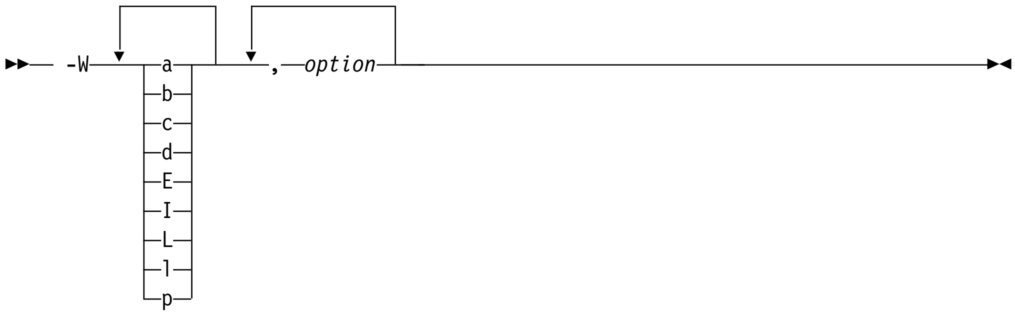
Pragma equivalent

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.

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

Parameter	Description	Component name
a	The assembler	as
b	The low-level optimizer	xlCcode
c	The compiler front end	xlcentry
d	The disassembler	dis
E	The CreateExportList utility	CreateExportList
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

In the string following the **-W** option, use a comma as the separator for each option, and do not include any spaces. If you need to include a character that is special to the shell in the option string, precede the character with a backslash. For example, if you use the **-W** option in the configuration file, you can use the escape sequence backslash comma (\,) to represent a comma in the parameter string.

You do not need the **-W** option to pass most options to the linker **ld**; unrecognized command-line options, except **-q** options, are passed to it automatically. Only linker options with the same letters as compiler options, such as **-v** or **-S**, strictly require **-W**.

Predefined macros

None.

Examples

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

```
xlc -Wl,-berok file.c
```

To compile the file `uses_many_symbols.c` and the assembly file `produces_warnings.s` so that `produces_warnings.s` is assembled with the assembler option **-x** (issue warnings and produce cross-reference), and the object files are linked with the option **-s** (write list of object files and strip final executable file), issue the following command:

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

Related information

- "Invoking the compiler" on page 1

-qwarn64

Category

Error checking and debugging

Pragma equivalent

None.

Purpose

Enables checking for possible data conversion problems between 32-bit and 64-bit compiler modes.

When **-qwarn64** is in effect, informational messages are displayed where data conversion may cause problems in 64-bit compilation mode, such as:

- Truncation due to explicit or implicit conversion of `long` types into `int` types
- Unexpected results due to explicit or implicit conversion of `int` types into `long` types
- Invalid memory references due to explicit conversion by cast operations of pointer types into `int` types

- Invalid memory references due to explicit conversion by cast operations of `int` types into pointer types
- Problems due to explicit or implicit conversion of constants into `long` types
- Problems due to explicit or implicit conversion by cast operations of constants into pointer types

Syntax

```
         ┌─nowarn64─┐
►►── -q──┴─warn64───┴──────────────────────────────────────────────►◄
```

Defaults

-qnowarn64

Usage

This option functions in either 32-bit or 64-bit compiler modes. In 32-bit mode, it functions as a preview aid to discover possible 32-bit to 64-bit migration problems.

Predefined macros

None.

Related information

- "-q32, -q64" on page 94
- "Compiler messages" on page 16

-qweakexp

Category

Object code control

Pragma equivalent

None.

Purpose

When used with the **-qmkshrobj** or **-G** option, includes or excludes global symbols marked as weak from the export list generated when you create a shared object.

Syntax

```
         ┌─weakexp───┐
►►── -q──┴─noweakexp─┴─────────────────────────────────────────────►◄
```

Defaults

-qweakexp: weak symbols are exported.

Usage

See “-qweaksymbol” for a description of weak symbols.

Use the **-qweakexp** option with the **-qmkshrobj** or **-G** option. See the description of “-qmkshrobj” on page 233 or “-G” on page 163 for more information.

Predefined macros

None.

Examples

To compile `myprogram.c` into a shared object and prevent weak symbols from being exported, enter the following command:

```
xlc myprogram.c -qmkshrobj -qnoweakexp
```

Related information

- “-qweaksymbol”
- “#pragma weak” on page 377
- “-qmkshrobj” on page 233
- “-G” on page 163

-qweaksymbol

Category

Object code control

Pragma equivalent

None.

Purpose

Enables the generation of weak symbols.

When the **-qweaksymbol** option is in effect, the compiler generates weak symbols for the following cases:

- Inline functions with external linkage
- Identifiers specified as weak with **#pragma weak** or `__attribute__((weak))`

Syntax

```
           ┌─weaksymbol───┐
►►── -q────┴─noweaksymbol─┴─────────────────────────────────────────────►◄
```

Defaults

-qweaksymbol

Predefined macros

None.

Related information

- “#pragma weak” on page 377
- “-qweakexp” on page 328
- "The weak variable attribute" and "The weak function attribute" in the *XL C Language Reference*

-qxcall

Category

Object code control

Pragma equivalent

None.

Purpose

Generates code to treat static functions within a compilation unit as if they were external functions.

Syntax

```
           ┌─noxcall─┐
►►── -q────┴─xcall───┴──────────────────────────────────────────────►◄
```

Defaults

-qnoxcall

Usage

-qxcall generates slower code than **-qnoxcall**.

Predefined macros

None.

Examples

To compile `myprogram.c` so that all static functions are compiled as external functions, enter:

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

-qxref

Category

Listings, messages, and compiler information

Pragma equivalent

#pragma options [no]xref

Purpose

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

When **xref** is in effect, a listing file is generated with a .lst suffix for each source file named on the command line. For details of the contents of the listing file, see "Compiler listings" on page 19.

Syntax

```
           ┌─noxref──────────┐
►►── -q ───┴─xref──┬────────┬┴──────────────────────────────────────►◄
                   └─=──full─┘
```

Defaults

-qnoxref

Parameters

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

Usage

A typical cross-reference listing has the form:

```
Identifier name      Description of the item
  ┌─────────┐       ┌─────────────────────────┐
      xy             auto int in function adder
                     0-59Y  0-36.12Z   0-48.12Z
                       │               │ ││ ││└─ Function invocation
                       │               │ ││ │└── Column number
                       │               │ │└──── Line number
                       │               └────── File
                       └──────────────────── Function definition
```

The listing uses the following character codes:

Table 26. Cross-reference listing codes

Character	Meaning
X	Function is declared.
Y	Function is defined.
Z	Function is called.
$	Type is defined, variable is declared/defined.
#	Variable is assigned to.
&	Variable is defined and initialized.
[blank]	Identifier is referenced.
{ and }	Coordinates of the { and } symbols in a structure definition.

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

Any function defined with the **#pragma mc_func** directive is listed as being defined on the line of the pragma directive.

Predefined macros

None.

Examples

To compile `myprogram.c` and produce a cross-reference listing of all identifiers, whether they are used or not, enter:

```
xlc myprogram.c -qxref=full
```

Related information

- "-qattr" on page 108
- "#pragma mc_func" on page 359

-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

```
         ┌─dn──┐
         ├─n───┤
►►── -y──┼─m───┼─────────────────────────────────────────────────►◄
         ├─p───┤
         ├─z───┤
         ├─di──┤
         ├─dm──┤
         ├─dna─┤
         ├─dnz─┤
         ├─dp──┤
         └─dz──┘
```

Defaults

- **-yn**
- **-ydn**

Parameters

The following suboptions are valid for binary floating-point types only:

m Round toward minus infinity.

n Round to the nearest representable number, ties to even.

p Round toward plus infinity.

z Round toward zero.

The following suboptions are valid for decimal floating-point types only:

di Round toward infinities (away from zero).

dm Round toward minus infinity.

dn Round to the nearest representable number, ties to even.

dna
Round to the nearest representable number, ties away from zero.

dnz
Round to the nearest representable number, ties toward zero.

dp Round toward plus infinity.

dz 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 -ydz
```

-Z

Category

Linking

Pragma equivalent

None.

Purpose

Specifies a prefix for the library search path to be used by the linker.

Syntax

```
►►— -Z—string————————————————————————————————————————————————►◄
```

Defaults

By default, the linker searches the /usr/lib/ directory for library files.

Parameters

string
Represents the prefix to be added to the directory search path for library files.

Predefined macros

None.

Chapter 5. Compiler pragmas reference

The following sections describe the available pragmas:

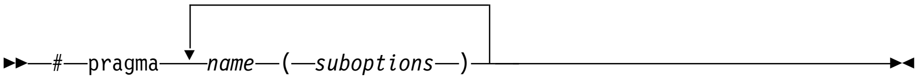
- "Pragma directive syntax"
- "Scope of pragma directives" on page 336
- "Summary of compiler pragmas by functional category" on page 336
- "Individual pragma descriptions" on page 339

Pragma directive syntax

XL C supports the following forms of pragma directives:

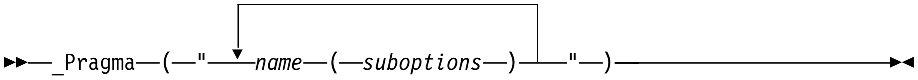
#pragma options *option_name*
These pragmas use exactly the same syntax as their command-line option equivalent. The exact syntax and list of supported pragmas of this type are provided in "#pragma options" on page 362.

#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 are included. For a directive that can appear anywhere in your source code, it applies from the point at which it is specified, until the end of the compilation unit.

You can further restrict the scope of a pragma's application by using complementary pairs of pragma directives around a selected section of code.

For example, using **#pragma options source** and **#pragma options nosource** directives as follows requests that only the selected parts of your source code be included in your compiler listing:

```
#pragma options source

/*  Source code between the source and nosource pragma
    options is included in the compiler listing              */

#pragma options nosource
```

Many pragmas provide "pop" or "reset" suboptions that allow you to enable and disable pragma settings in a stack-based fashion; examples of these are provided in the relevant pragma descriptions.

Summary of compiler pragmas by functional category

The XL C pragmas available are grouped into the following categories:
- “Language element control”
- “Floating-point and integer control” on page 337
- “Error checking and debugging” on page 337
- “Optimization and tuning” on page 337
- “Object code control” on page 338
- “Portability and migration” on page 339
- “Deprecated directives” on page 339

For descriptions of these categories, see “Summary of compiler options by functional category” on page 75.

Language element control

Table 27. Language element control pragmas

Pragma	Description
#pragma langlvl (C only)	Determines whether source code and compiler options should be checked for conformance to a specific language standard, or subset or superset of a standard.
“#pragma mc_func” on page 359	Allows you to embed a short sequence of machine instructions "inline" within your program source code.

Table 27. Language element control pragmas (continued)

Pragma	Description
"#pragma options" on page 362	Specifies compiler options in your source program.

Floating-point and integer control

Table 28. Floating-point and integer control pragmas

Pragma	Description
#pragma chars	Determines whether all variables of type `char` is treated as `signed` or `unsigned`.
#pragma enum	Specifies the amount of storage occupied by enumerations.

Error checking and debugging

Table 29. Error checking and debugging pragmas

Pragma	Description
"#pragma ibm snapshot" on page 355	Specifies a location at which a breakpoint can be set and defines a list of variables that can be examined when program execution reaches that location.
#pragma info	Produces or suppresses groups of informational messages.

Optimization and tuning

Table 30. Optimization and tuning pragmas

Pragma	Description
"#pragma block_loop" on page 340	Marks a block with a scope-unique identifier.
"#pragma STDC CX_LIMITED_RANGE" on page 373	Informs the compiler that complex division and absolute value are only invoked with values such that intermediate calculation will not overflow or lose significance.
"#pragma disjoint" on page 345	Lists identifiers that are not aliased to each other within the scope of their use.
"#pragma execution_frequency" on page 346	Marks program source code that you expect will be either very frequently or very infrequently executed.
"#pragma expected_value" on page 348	Specifies the value that a parameter passed in a function call is most likely to take at run time. The compiler can use this information to perform certain optimizations, such as function cloning and inlining.
"#pragma GCC visibility push, #pragma GCC visibility pop" on page 349	Specifies the visibility attribute for external linkage entities in object files.
"#pragma ibm iterations" on page 352	Specifies the approximate average number of loop iterations for the chosen loop.
"#pragma ibm max_iterations" on page 353	Specifies the approximate maximum number of loop iterations for the chosen loop.

Table 30. Optimization and tuning pragmas (continued)

Pragma	Description
"#pragma ibm min_iterations" on page 354	Specifies the approximate minimum number of loop iterations for the chosen loop.
#pragma isolated_call	Specifies functions in the source file that have no side effects other than those implied by their parameters.
"#pragma leaves" on page 356	Informs the compiler that a named function never returns to the instruction following a call to that function.
"#pragma loopid" on page 357	Marks a block with a scope-unique identifier.
#pragma nosimd	When used with **-qsimd=auto**, disables the generation of SIMD instructions for the next loop.
#pragma novector	When used with **-qhot=vector**, disables auto-vectorization of the next loop.
"#pragma option_override" on page 364	Allows you to specify optimization options at the subprogram level that override optimization options given on the command line.
"#pragma reachable" on page 369	Informs the compiler that the point in the program after a named function can be the target of a branch from some unknown location.
"#pragma reg_killed_by" on page 370	Specifies registers that may be altered by functions specified by **#pragma mc_func**.
"#pragma simd_level" on page 372	Controls the compiler code generation of vector instructions for individual loops.
"#pragma stream_unroll" on page 374	When optimization is enabled, breaks a stream contained in a `for` loop into multiple streams.
#pragma unroll	Controls loop unrolling, for improved performance.
"#pragma unrollandfuse" on page 375	Instructs the compiler to attempt an unroll and fuse operation on nested `for` loops.

Object code control

Table 31. Object code control pragmas

Pragma	Description
#pragma alloca (C only)	Provides an inline definition of system function `alloca` when it is called from source code that does not include the `alloca.h` header.
"#pragma comment" on page 343	Places a comment into the object module.
"#pragma fini" on page 349	Specifies the order in which the runtime library calls a list of functions after main() completes or exit() is called.

Table 31. Object code control pragmas (continued)

Pragma	Description
“#pragma init” on page 355	Specifies the order in which the runtime library calls a list of functions before main() is called.
“#pragma map” on page 358	Converts all references to an identifier to another, externally defined identifier.
“#pragma pack” on page 366	Sets the alignment of all aggregate members to a specified byte boundary.
“#pragma reg_killed_by” on page 370	Specifies registers that may be altered by functions specified by **#pragma mc_func**.
#pragma strings	Specifies the storage type for string literals.
“#pragma weak” on page 377	Prevents the linker from issuing error messages if it encounters a symbol multiply-defined during linking, or if it does not find a definition for a symbol.

Portability and migration

Table 32. Portability and migration pragmas

Pragma	Description
#pragma align	Specifies the alignment of data objects in storage, which avoids performance problems with misaligned data.

Deprecated directives

The SMP directive listed in the following table has been deprecated and might be removed in a future release. Use the corresponding OpenMP directive to obtain the same behavior.

Table 33. Deprecated SMP directives

SMP directive name	OpenMP directive/clause name
#pragma ibm schedule	The “#pragma omp parallel for” on page 393 pragma with the**schedule** clause.

You can replace the deprecated SMP directive with the corresponding OpenMP one. For example:

```
#pragma omp parallel for schedule(static, 5)
for (i=0; i<N; i++)
{
  // ...
}
```

Individual pragma descriptions

This section contains descriptions of individual pragmas available in XL 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 335 for details.

Parameters
This section describes the suboptions that are available for the pragma, where applicable.

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

Examples
Where appropriate, examples of pragma directive use are provided in this section.

#pragma align

See "-qalign" on page 98.

#pragma alloca

See "-qalloca, -ma" on page 100.

#pragma block_loop

Category

Optimization and tuning

Purpose

Marks a block with a scope-unique identifier.

Syntax

```
                                                    ┌─,────┐
                                                    ▼      │
►►─#─pragma─block_loop─(─expression─,───name─┴─)─────────────────────►◄
```

Parameters

expression
An integer expression representing the size of the iteration group.

name
An identifier that is unique within the scoping unit. If you do not specify a *name*, blocking occurs on the first `for` loop or loop following the **#pragma block_loop** directive.

Usage

For loop blocking to occur, a **#pragma block_loop** directive must precede a `for` loop.

If you specify **#pragma unroll**, **#pragma unrollandfuse** or **#pragma stream_unroll** for a blocking loop, the blocking loop is unrolled, unrolled and fused or stream unrolled respectively, if the blocking loop is actually created. Otherwise, this directive has no effect.

If you specify **#pragma unrollandfuse**, **#pragma unroll** or **#pragma stream_unroll** directive for a blocked loop, the directive is applied to the blocked loop after the blocking loop is created. If the blocking loop is not created, this directive is applied to the loop intended for blocking, as if the corresponding **#pragma block_loop** directive was not specified.

You must not specify **#pragma block_loop** more than once, or combine the directive with **#pragma nounroll**, **#pragma unroll**, **#pragma nounrollandfuse**, **#pragma unrollandfuse**, or **#pragma stream_unroll** directives for the same `for` loop. Also, you should not apply more than one **#pragma unroll** directive to a single block loop directive.

Processing of all **#pragma block_loop** directives is always completed before performing any unrolling indicated by any of the unroll directives

Examples

The following two examples show the use of **#pragma block_loop** and **#pragma loop_id** for loop tiling:

```
#pragma block_loop(50, mymainloop)
#pragma block_loop(20, myfirstloop, mysecondloop)
#pragma loopid(mymainloop)
  for (i=0; i < n; i++)
  {
#pragma loopid(myfirstloop)
    for (j=0; j < m; j++)
    {
#pragma loopid(mysecondloop)
      for (k=0; k < m; k++)
      {
         ...
      }
    }
  }
```

```
#pragma block_loop(50, mymainloop)
#pragma block_loop(20, myfirstloop, mysecondloop)
#pragma loopid(mymainloop)
       for (i=0; i < n; n++)
       {
#pragma loopid(myfirstloop)
              for (j=0; j < m; j++)
              {
#pragma loopid(mysecondloop)
                     for (k=0; k < m; k++)
                     {
                            ...
                     }
              }
       }
```

The following example shows the use **#pragma block_loop** and **#pragma loop_id** for loop interchange.

```
       for (i=0; i < n; i++)
       {
              for (j=0; j < n; j++)
              {
#pragma block_loop(1,myloop1)
                     for (k=0; k < m; k++)
                     {
#pragma loopid(myloop1)
                            for (l=0; l < m; l++)
                            {
                                   ...
                            }
                     }
              }
       }
```

The following example shows the use of **#pragma block_loop** and **#pragma loop_id** for loop tiling for multi-level memory hierarchy:

```
 #pragma block_loop(l3factor, first_level_blocking)
   for (i=0; i < n; i++)
   {
 #pragma loopid(first_level_blocking)
 #pragma block_loop(l2factor, inner_space)
     for (j=0; j < n; j++)
     {
 #pragma loopid(inner_space)
       for (k=0; k < m; k++)
       {
         for (l=0; l < m; l++)
         {
           ...
         }
       }
     }
   }
```

The following example uses **#pragma unrollandfuse** and **#pragma block_loop** to unroll and fuse a blocking loop.

```
#pragma unrollandfuse
#pragma block_loop(10)
   for (i = 0; i < N; ++i) {
   }
```

In this case, if the block loop directive is ignored, the unroll directives have no effect.

The following example shows the use of **#pragma unroll** and **#pragma block_loop** to unroll a blocked loop.

```
 #pragma block_loop(10)
 #pragma unroll(2)
   for (i = 0; i < N; ++i) {
   }
```

In this case, if the block loop directive is ignored, the unblocked loop is still subjected to unrolling. If blocking does happen, the unroll directive is applied to the blocked loop.

The following examples show invalid uses of the directive. The first example shows **#pragma block_loop** used on an undefined loop identifier:

```
#pragma block_loop(50, myloop)
  for (i=0; i < n; i++)
  {
  }
```

Referencing myloop is not allowed, since it is not in the nest and may not be defined.

In the following example, referencing myloop is not allowed, since it is not in the same loop nest:

```
  for (i=0; i < n; i++)
  {
#pragma loopid(myLoop)
    for (j=0; j < i; j++)
    {
      ...
    }
  }
#pragma block_loop(myLoop)
  for (i=0; i < n; i++)
  {
    ...
  }
```

The following examples are invalid since the unroll directives conflict with each other:

```
#pragma unrollandfuse(5)
#pragma unroll(2)
  #pragma block_loop(10)
           for (i = 0; i < N; ++i) {
       }
#pragma block_loop(10)
#pragma unroll(5)
#pragma unroll(10)
  for (i = 0; i < N; ++i) {
  }
```

Related information

- “#pragma loopid” on page 357
- “-qunroll” on page 314
- “#pragma unrollandfuse” on page 375
- “#pragma stream_unroll” on page 374

#pragma chars

See “-qchars” on page 118.

#pragma comment

Category

Object code control

Purpose

Places a comment into the object module.

Syntax

```
►►─#─pragma─comment─(─┬─compiler──────────────────────────────────┬─)─────►◄
                      ├─date──────────────────────────────────────┤
                      ├─timestamp─────────────────────────────────┤
                      └─┬─copyright─┬─┬─────────────────────────┬─┘
                        └─user──────┘ └─,─"─token_sequence─"─┘
```

Parameters

compiler
Appends the name and version of the compiler at the end of the generated object module.

date
The date and time of the compilation are appended at the end of the generated object module.

timestamp
Appends the date and time of the last modification of the source at the end of the generated object module.

copyright
Places the text specified by the *token_sequence*, if any, into the generated object module. The *token_sequence* is included in the generated executable and loaded into memory when the program is run.

user
Places the text specified by the *token_sequence*, if any, into the generated object module. The *token_sequence* is included in the generated executable but is *not* loaded into memory when the program is run.

token_sequence
The characters in this field, if specified, must be enclosed in double quotation marks ("). If the string literal specified in the *token_sequence* exceeds 32 767 bytes, an information message is emitted and the pragma is ignored.

Usage

More than one **comment** directive can appear in a translation unit, and each type of **comment** directive can appear more than once, with the exception of **copyright**, which can appear only once.

You can display the object-file comments by using the operating system **strings** command.

Examples

Assume that the code of `tt.c` is as follows:

```
#pragma comment(date)
#pragma comment(compiler)
#pragma comment(timestamp)
#pragma comment(copyright,"My copyright")
int main() { return 0; }
```

To display the comment information embedded in `tt.o`, along with any other strings that can be found in the code, issue the command:

```
xlc -c tt.c
strings -a tt.o
```

The preceding code might produce the following results:

```
@.text
.data
@.bss
.comment
Thu Dec 24 16:44:25 EDT 2015IBM XL C for AIX ---- Version 13.1.3.0
Thu Dec 24 16:44:09 EDT 2015
main
My copyright
.file
tt.c
.text
.data
.bss
.main
_$STATIC
_$STATIC
main
main
Thu Dec 24 16:44:25 2015
IBM XL C for AIX, Version 13.1.3.0 ---
```

#pragma disjoint

Category

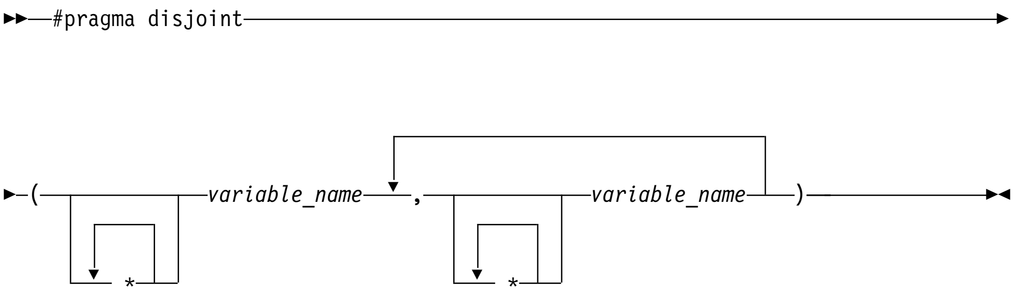
Optimization and tuning

Purpose

Lists identifiers that are not aliased to each other within the scope of their use.

By informing the compiler that none of the identifiers listed in the pragma shares the same physical storage, the pragma provides more opportunity for optimizations.

Syntax



Parameters

variable_name
The name of a variable. It must not refer to any of the following:

- A member of a structure or union
- A structure, union, or enumeration tag
- An enumeration constant
- A `typedef` name
- A label

Usage

The **#pragma disjoint** directive asserts that none of the identifiers listed in the pragma share physical storage; if any the identifiers *do* actually share physical storage, the pragma may give incorrect results.

The pragma can appear anywhere in the source program that a declaration is allowed. An identifier in the directive must be visible at the point in the program where the pragma appears.

You must declare the identifiers before using them in the pragma. Your program must not dereference a pointer in the identifier list nor use it as a function argument before it appears in the directive.

This pragma can be disabled with the **-qignprag** compiler option.

Examples

The following example shows the use of **#pragma disjoint**.

```
int a, b, *ptr_a, *ptr_b;

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 enum

See “-qenum” on page 137.

#pragma execution_frequency

Category

Optimization and tuning

Purpose

Marks program source code that you expect will be either very frequently or very infrequently executed.

When optimization is enabled, the pragma is used as a hint to the optimizer.

Syntax

```
►►──#──pragma──execution_frequency──(──┬─very_low──┬──)──────────────────────────►◄
                                       └─very_high─┘
```

Parameters

very_low
Marks source code that you expect will be executed very infrequently.

very_high
Marks source code that you expect will be executed very frequently.

Usage

Use this pragma in conjunction with an optimization option; if optimization is not enabled, the pragma has no effect.

The pragma must be placed within block scope, and acts on the closest 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 expected_value

Category

Optimization and tuning

Purpose

Specifies the value that a parameter passed in a function call is most likely to take at run time. The compiler can use this information to perform certain optimizations, such as function cloning and inlining.

Syntax

```
►►—#pragma expected_value—(—argument—,—value—)——————————————————————————►◄
```

Parameters

argument
: The name of the parameter for which you want to provide the expected value. The parameter must be of a simple built-in integral, Boolean, character, or floating-point type.

value
: A constant literal representing the value that you expect will most likely be taken by the parameter at run time. *value* can be an expression as long as it is a compile time constant expression.

Usage

The directive must appear inside the body of a function definition, before the first statement (including declaration statements). It is not supported within nested functions.

If you specify an expected value of a type different from that of the declared type of the parameter variable, the value will be implicitly converted only if allowed. Otherwise, a warning is issued.

For each parameter that will be provided the expected value there is a limit of one directive. Parameters that will not be provided the expected value do not require a directive.

Examples

The following example tells the compiler that the most likely values for parameters a and b are 1 and 0, respectively:

```
int func(int a,int b)
{
#pragma expected_value(a,1)
#pragma expected_value(b,0)
...
...
}
```

Related information

- “#pragma execution_frequency” on page 346

#pragma fini

Category

“Object code control” on page 338

Purpose

Specifies the order in which the runtime library calls a list of functions after main() completes or exit() is called.

For shared libraries, the fini functions are called when the shared library is loaded from memory. For example, when using dynamic loading, this happens at the point when dlclose() is called.

Syntax

```
                          ┌─,─────────────┐
                          ▼               │
►►──#──pragma──fini──(────function_name───┴──)──────────────────────────►◄
```

Usage

Any function that is specified in the pragma should have return type void (for example, `void fA();`) and take no parameters. Functions that have a non-void return type are accepted but the return value is discarded.

Functions that take parameters are ignored with a warning since the parameters would contain garbage values.

Within the same compilation unit, the list of functions in `pragma fini` are called in the order specified. Similarly, within the same compilation unit, functions specified in more than one pragma fini are called in the order in which the pragmas are encountered in the source.

In general, the order of static termination across files and across libraries is nonstandard and therefore, a non-portable behavior. It is not advisable to build any dependency on this behavior. The order of functions across files is undefined, even when using the **-Wm** option.

When mixing C and C++ files, the relative order of init or fini functions in C files with respect to the static constructors/destructors in C++ files is undefined. The **-qunique** option can interact with `pragma fini`.

Related information

- “#pragma init” on page 355

#pragma GCC visibility push, #pragma GCC visibility pop

Category

Optimization and tuning

Purpose

Specifies the visibility attribute for external linkage entities in object files.

Syntax

```
►►─#─pragma─GCC─visibility─push─(──┬─default───┬─)──────────────────────►◄
                                   ├─protected─┤
                                   ├─hidden────┤
                                   └─internal──┘
```

```
►►─#─pragma─GCC─visibility─pop──────────────────────────────────────────►◄
```

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.

internal
Indicates that the affected external linkage entities have the internal visibility attribute. These entities are not exported in shared libraries, and their addresses are not available to other modules.

Restriction: In this release, the hidden and internal visibility attributes are the same. The addresses of the entities that are specified with either of these visibility attributes can be referenced indirectly through pointers.

Usage

You can selectively set visibility attributes for entities by using pairs of the `#pragma GCC visibility push` and `#pragma GCC visibility pop` compiler directives throughout your source program. If you specify the `#pragma GCC visibility pop` directive without the corresponding `#pragma GCC visibility push` directive, the compiler issues a warning message. Entity visibility attributes describe whether and how an entity defined in one module can be referenced or used in other modules. 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.

Note: On the AIX platform, entity preemption occurs only when runtime linking is used. For details, see "Linking a library to an application" in the *XL C Optimization and Programming Guide*. Visibility attributes are supported on AIX 6.1 TL8, AIX 7.1 TL2, AIX 7.2, and higher.

Related information

- “-qvisibility” on page 323
- “-qmkshrobj” on page 233
- “-G” on page 163
- "Using visibility attributes (IBM extension)" in the *XL C Optimization and Programming Guide*
- "External linkage", "The visibility variable attribute (IBM extension)", and "The visibility function attribute (IBM extension)" in the *XL C Language Reference*

#pragma ibm independent_loop

Purpose

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

Syntax

```
►►—#—pragma—ibm independent_loop—┬────────┬————————————————————————►◄
                                  └if exp─┘
```

where exp represents a scalar expression.

Usage

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

Notes:

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

This pragma can be combined with the **omp parallel for** pragma to select a specific parallel process scheduling algorithm. For more information, see “#pragma omp parallel for” on page 393.

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 ibm iterations

Category

Optimization and tuning

Purpose

The **iterations** pragma specifies the approximate average number of loop iterations for the chosen loop.

Syntax

```
►►──#──pragma──ibm iterations──(iteration_count)──────────────────────────►◄
```

Parameters

iteration_count
Specifies the approximate number of loop iterations using a positive integral constant expression.

Usage

The compiler uses the information in *iteration_count* for loop optimization. You can specify multiple #pragma ibm iterations(*iteration_count*).

iteration_count specified in #pragma ibm iterations cannot be smaller than *iteration_count* specified in #pragma ibm min_iterations. In addition, it cannot be bigger than *iteration_count* specified in #pragma ibm max_iterations. Otherwise, the inconsistent value is ignored with a message.

Example

```
#pragma ibm     iterations(100)        // Accepted
#pragma ibm min_iterations(150)        // Ignored (150 > 100)
#pragma ibm min_iterations( 30)        // Accepted( 30 < 100)
#pragma ibm max_iterations( 60)        // Ignored ( 60 < 100)
#pragma ibm     iterations( 20)        // Ignored ( 20 < 30)
#pragma ibm max_iterations(500)        // Accepted(500 > 100 > 30)
#pragma ibm max_iterations(620)        // Ignored (Multiple occurrences)
#pragma ibm     iterations(200)        // Accepted( 30 < 200 < 500)
#pragma ibm min_iterations( 15)        // Ignored (Multiple occurrences)

   for (int i=0; i < n; ++i)
   {
     #pragma ibm max_iterations(130)   // Accepted
     #pragma ibm min_iterations( 90)   // Accepted( 90 < 130)
     #pragma ibm     iterations( 60)   // Ignored ( 60 <  90)
     #pragma ibm     iterations(100)   // Accepted( 90 < 100 < 130)

     for (int j=0; j < m; ++j) b[j] += a[i];
   }
```

Related reference:

"#pragma ibm max_iterations" on page 353

"#pragma ibm min_iterations" on page 354

#pragma ibm max_iterations

Category

Optimization and tuning

Purpose

The **max_iterations** pragma specifies the approximate maximum number of loop iterations for the chosen loop.

Syntax

```
►►—#—pragma—ibm max_iterations—(iteration_count)——————————————————————————►◄
```

Parameters

iteration_count
Specifies the approximate number of maximum loop iterations using a positive integral constant expression.

Usage

The compiler uses the information in *iteration_count* for loop optimization. You can specify #pragma ibm max_iterations(*iteration_count*) only once. If you specify #pragma ibm max_iterations(*iteration_count*) more than once, the first specified pragma is accepted, and the subsequent pragmas are ignored with a message.

iteration_count specified in #pragma ibm max_iterations cannot be smaller than *iteration_count* specified in #pragma ibm iterations or #pragma ibm min_iterations. Otherwise, the inconsistent value is ignored with a message.

Example

```
#pragma ibm     iterations(100)       // Accepted
#pragma ibm min_iterations(150)       // Ignored (150 > 100)
#pragma ibm min_iterations( 30)       // Accepted( 30 < 100)
#pragma ibm max_iterations( 60)       // Ignored ( 60 < 100)
#pragma ibm     iterations( 20)       // Ignored ( 20 < 30)
#pragma ibm max_iterations(500)       // Accepted(500 > 100 > 30)
#pragma ibm max_iterations(620)       // Ignored (Multiple occurrences)
#pragma ibm     iterations(200)       // Accepted( 30 < 200 < 500)
#pragma ibm min_iterations( 15)       // Ignored (Multiple occurrences)

   for (int i=0; i < n; ++i)
   {
     #pragma ibm max_iterations(130)   // Accepted
     #pragma ibm min_iterations( 90)   // Accepted( 90 < 130)
     #pragma ibm     iterations( 60)   // Ignored ( 60 <  90)
     #pragma ibm     iterations(100)   // Accepted( 90 < 100 < 130)

     for (int j=0; j < m; ++j) b[j] += a[i];
   }
```

Related reference:

"#pragma ibm iterations" on page 352

"#pragma ibm min_iterations" on page 354

#pragma ibm min_iterations

Category

Optimization and tuning

Purpose

The **min_iterations** pragma specifies the approximate minimum number of loop iterations for the chosen loop.

Syntax

```
►►—#—pragma—ibm min_iterations—(iteration_count)——————————————————————————►◄
```

Parameters

iteration_count
Specifies the approximate minimum number of loop iterations using a positive integral constant expression.

Usage

The compiler uses the information in *iteration_count* for loop optimization. You can specify #pragma ibm min_iterations(*iteration_count*) only once. If you specify #pragma ibm min_iterations(*iteration_count*) more than once, the first specified pragma is accepted, and the subsequent pragmas are ignored with a message.

iteration_count specified in #pragma ibm min_iterations cannot be bigger than *iteration_count* specified in #pragma ibm iterations or #pragma ibm max_iterations. Otherwise, the inconsistent value is ignored with a message.

Example

```
#pragma ibm     iterations(100)        // Accepted
#pragma ibm min_iterations(150)        // Ignored (150 > 100)
#pragma ibm min_iterations( 30)        // Accepted( 30 < 100)
#pragma ibm max_iterations( 60)        // Ignored ( 60 < 100)
#pragma ibm     iterations( 20)        // Ignored ( 20 < 30)
#pragma ibm max_iterations(500)        // Accepted(500 > 100 > 30)
#pragma ibm max_iterations(620)        // Ignored (Multiple occurrences)
#pragma ibm     iterations(200)        // Accepted( 30 < 200 < 500)
#pragma ibm min_iterations( 15)        // Ignored (Multiple occurrences)

   for (int i=0; i < n; ++i)
   {
     #pragma ibm max_iterations(130)   // Accepted
     #pragma ibm min_iterations( 90)   // Accepted( 90 < 130)
     #pragma ibm     iterations( 60)   // Ignored ( 60 <  90)
     #pragma ibm     iterations(100)   // Accepted( 90 < 100 < 130)

     for (int j=0; j < m; ++j) b[j] += a[i];
   }
```

Related reference:

"#pragma ibm iterations" on page 352

"#pragma ibm max_iterations" on page 353

#pragma ibm snapshot

Category

Error checking and debugging

Purpose

Specifies a location at which a breakpoint can be set and defines a list of variables that can be examined when program execution reaches that location.

You can use this pragma to facilitate debugging optimized code produced by the compiler.

Syntax

```
                                   ┌─,──────────────┐
                                   ▼                │
►►──#──pragma──ibm snapshot──(────variable_name────┴──)──────────────────────►◄
```

Parameters

variable_name
A variable name. It must not refer to structure or union members.

Usage

During a debugging session, you can place a breakpoint on the line at which the directive appears, to view the values of the named variables. When you compile with optimization and the **-g** option, the named variables are guaranteed to be visible to the debugger.

This pragma does not consistently preserve the contents of variables with a static storage class at high optimization levels. Variables specified in the directive should be considered read-only while being observed in the debugger, and should not be modified. Modifying these variables in the debugger may result in unpredictable behavior.

Examples

```
#pragma ibm snapshot(a, b, c)
```

Related information

- “-g” on page 160
- “-O, -qoptimize” on page 236

#pragma info

See “-qinfo” on page 178.

#pragma init

Category

“Object code control” on page 338

Purpose

Specifies the order in which the runtime library calls a list of functions before main() is called.

For shared libraries, the init functions are called when the shared library is loaded to memory. For example, when using dynamic loading, this happens at the point when dlopen() is called.

Syntax

```
                      ┌─,───────────┐
                      ▼             │
►►──#──pragma──init──(───function_name─┴──)──────────────────────────────►◄
```

Usage

Any function that is specified in the pragma should have return type void (for example, `void fA();`) and take no parameters. Functions that have a non-void return type are accepted but the return value is discarded.

Functions that take parameters are ignored with a warning since the parameters would contain garbage values.

Within the same compilation unit, the list of functions in `pragma init` are called in the order specified. Similarly, within the same compilation unit, functions specified in more than one `pragma init` are called in the order in which the pragmas are encountered in the source.

In general, the order of static initialization across files and across libraries is nonstandard and therefore, a non-portable behavior. It is not advisable to build any dependency on this behavior. The order of functions across files is undefined, even when using the **-Wm** option).

Related information

- "#pragma fini" on page 349
-

#pragma isolated_call

See "-qisolated_call" on page 199.

#pragma langlvl

See "-qlanglvl" on page 206.

#pragma leaves

Category

Optimization and tuning

Purpose

Informs the compiler that a named function never returns to the instruction following a call to that function.

By informing the compiler that it can ignore any code after the function, the directive allows for additional opportunities for optimization.

This pragma is commonly used for custom error-handling functions, in which programs can be terminated if a certain error is encountered.

Note: The compiler automatically inserts **#pragma leaves** directives for calls to the `longjmp` family of functions (`longjmp`, `_longjmp`, `siglongjmp`, and `_siglongjmp`) when you include the setjmp.h header.

Syntax

```
                          ┌─,─────────────┐
                          ▼               │
►►──#──pragma──leaves──(────function_name─┴──)──────────────────────────────►◄
```

Parameters

function_name
: The name of the function that does not return to the instruction following the call to it.

Defaults

Not applicable.

Examples

```
#pragma leaves(handle_error_and_quit)
void test_value(int value)
{
 if (value == ERROR_VALUE)
 {
  handle_error_and_quit(value);
  TryAgain(); // optimizer ignores this because
    // never returns to execute it
 }
}
```

Related information

- “#pragma reachable” on page 369.

#pragma loopid

Category

Optimization and tuning

Purpose

Marks a block with a scope-unique identifier.

Syntax

```
►►──#──pragma──loopid──(──name──)────────────────────────────────────────────►◄
```

Parameters

name
An identifier that is unique within the scoping unit.

Usage

The **#pragma loopid** directive must immediately precede a **#pragma block_loop** directive or `for` loop. The specified name can be used by **#pragma block_loop** to control transformations on that loop. It can also be used to provide information on loop transformations through the use of the **-qreport** compiler option.

You must not specify **#pragma loopid** more than once for a given loop.

Examples

For examples of **#pragma loopid** usage, see "#pragma block_loop" on page 340.

Related information

- "-qunroll" on page 314
- "#pragma block_loop" on page 340
- "#pragma unrollandfuse" on page 375

#pragma map

Category

Object code control

Purpose

Converts all references to an identifier to another, externally defined identifier.

Syntax

#pragma map syntax (C only)

```
►►—#—pragma—map—(—name1—,—"—name2—"—)——————————————————————————————►◄
```

Parameters

name1
The name used in the source code. *name1* can represent a data object or function with external linkage.

name1 should be declared in the same compilation unit in which it is referenced, but should not be defined in any other compilation unit. *name1* must not be used in another **#pragma map** directive or any assembly label declaration anywhere in the program.

name2
The name that will appear in the object code. *name2* can represent a data object or function with external linkage.

If the name exceeds 65535 bytes, an informational message is emitted and the pragma is ignored.

name2 may or may not be declared in the same compilation unit in which *name1* is referenced, but must not be defined in the same compilation unit.

Also, *name2* should not be referenced anywhere in the compilation unit where *name1* is referenced. *name2* must not be the same as that used in another **#pragma map** directive or any assembly label declaration in the same compilation unit.

Usage

The **#pragma map** directive can appear anywhere in the program. Note that in order for a function to be actually mapped, the map target function (*name2*) must have a definition available at link time (from another compilation unit), and the map source function (*name1*) must be called in your program.

You cannot use **#pragma map** with compiler built-in functions.

Examples

The following is an example of **#pragma map** used to map a function name:

```
/* Compilation unit 1: */

#include <stdio.h>

void foo();
extern void bar(); /* optional */


#pragma map (foo, "bar")

int main()
{
foo();
}

/* Compilation unit 2: */

#include <stdio.h>

void bar()
{
printf("Hello from foo bar!\n");
}
```

The call to `foo` in compilation unit 1 resolves to a call to `bar`:

```
Hello from foo bar!
```

Related information

- "Assembly labels" in the *XL C Language Reference*

#pragma mc_func

Category

Language element control

Purpose

Allows you to embed a short sequence of machine instructions "inline" within your program source code.

The pragma instructs the compiler to generate specified instructions in place rather than the usual linkage code. Using this pragma avoids performance penalties associated with making a call to an assembler-coded external function. This pragma is similar in function to inline `asm` statements supported in this and other compilers; see "Inline assembly statements" in the *XL C Language Reference* for more information.

Syntax

```
                                           ┌─────────────────────┐
                                           ▼                     │
►►─#─pragma─mc_func─function_name─{────instruction_sequence─┴──}──────────►◄
```

Parameters

function_name
The name of a previously-defined function containing machine instructions. If the function is not previously-defined, the compiler will treat the pragma as a function definition.

instruction_sequence
A string containing a sequence of zero or more hexadecimal digits. The number of digits must comprise an integral multiple of 32 bits. If the string exceeds 16384 bytes, a warning message is emitted and the pragma is ignored.

Usage

This pragma defines a function and should appear in your program source only where functions are ordinarily defined.

The compiler passes parameters to the function in the same way as to any other function. For example, in functions taking integer-type arguments, the first parameter is passed to GPR3, the second to GPR4, and so on. Values returned by the function will be in GPR3 for integer values, and FPR1 for float or double values.

Code generated from *instruction_sequence* may use any and all volatile registers available on your system unless you use **#pragma reg_killed_by** to list a specific register set for use by the function. See "#pragma reg_killed_by" on page 370 for a list of volatile registers available on your system.

Inlining options do not affect functions defined by **#pragma mc_func**. However, you might improve runtime performance of such functions with **#pragma isolated_call**.

Examples

In the following example, **#pragma mc_func** is used to define a function called `add_logical`. The function consists of machine instructions to add 2 integers with so-called *end-around carry*; that is, if a carry out results from the add then add the carry to the sum. This formula is frequently used in checksum computations.

```
int add_logical(int, int);
#pragma mc_func add_logical {"7c632014" "7c630194"}
                /*   addc       r3 <- r3, r4           */
                /*   addze      r3 <- r3, carry bit    */


main() {
      int i,j,k;

      i = 4;
      k = -4;
      j = add_logical(i,k);
      printf("\n\nresult = %d\n\n",j);
}
```

The result of running the program is as follows:

```
result = 1
```

Related information

- "-qisolated_call" on page 199
- "#pragma reg_killed_by" on page 370
- "Inline assembly statements" in the *XL C Language Reference*

#pragma nofunctrace

Category

Error checking and debugging

Purpose

Disables tracing for a given function or a list of specified functions.

Syntax

```
                                   ┌─,─────────────┐
                                   ▼               │
▶▶─#─pragma─nofunctrace─(─────function_name─┴──)──────────────────────▶◀
```

Parameters

function_name
The name of the function for which you want to disable tracing.

Usage

When you use **#pragma nofunctrace** to specify a list of functions for which you want to disable tracing, use parenthesis **()** and encapsulate the functions in it. For a list of functions, use a comma **,** to separate them. For example, to disable tracing for function a, use #pragma nofunctrace(a). To disable tracing for functions a, b, and c, use #pragma nofunctrace(a,b,c).

Two colons in a row :: are considered scope qualifiers. For example, when you call -qfunctrace+A::B:C, the compiler traces functions that begin with the qualifiers A::B or C.

Note: If you want to use the compiler option **-qfunctrace** to disable tracing for a given function or a list of functions, you must use its suboption **-** followed by the names of the functions. For details about how to use **-qfunctrace** and its related suboptions, see "-qfunctrace" on page 158.

Examples

```
#pragma nofunctrace(a,b,c)
```

Related information

- "-qfunctrace" on page 158

#pragma nosimd

See "-qsimd" on page 278.

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 */
}
```

#pragma novector

See "-qhot" on page 169.

#pragma options

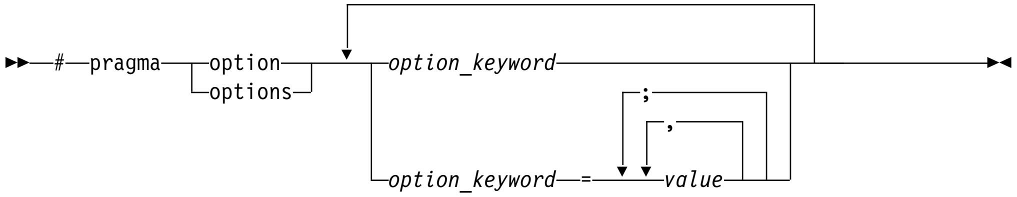
Category

Language element control

Purpose

Specifies compiler options in your source program.

Syntax



Parameters

The settings in the table below are valid *options* for **#pragma options**. For more information, see the pages of the equivalent compiler option.

Valid settings for #pragma options *option_keyword*	**Compiler option equivalent**
align=*option*	"-qalign" on page 98
[no]attr attr=full	"-qattr" on page 108
chars=*option*	"-qchars" on page 118
[no]check	"-qcheck" on page 119
[no]compact	"-qcompact" on page 122

Valid settings for #pragma options ***option_keyword***	**Compiler option equivalent**
[no]dbcs	“-qmbcs, -qdbcs” on page 230
[no]dbxextra	“-qdbxextra” on page 130
[no]digraph	“-qdigraph” on page 132
[no]dollar	“-qdollar” on page 133
enum=*option*	“-qenum” on page 137
[no]extchk	“-qextchk” on page 141
flag=*option*	“-qflag” on page 145
float=[no]*option*	“-qfloat” on page 146
[no]flttrap	“-qflttrap” on page 151
[no]fullpath	“-qfullpath” on page 156
halt	“-qhalt” on page 165
[no]idirfirst	“-qidirfirst” on page 173
[no]ignerrno	“-qignerrno” on page 174
ignprag=*option*	“-qignprag” on page 175
[no]info=*option*	“-qinfo” on page 178
initauto=*value*	“-qinitauto” on page 186
[no]inlglue	“-qinlglue” on page 188
isolated_call=*names*	“-qisolated_call” on page 199
langlvl	“-qlanglvl” on page 206
[no]ldbl128	“-qldbl128, -qlongdouble” on page 212
[no]libansi	“-qlibansi” on page 214
[no]list	“-qlist” on page 217
[no]longlong	“-qlonglong” on page 223
[no]macpstr	“-qmacpstr” on page 224
[no]maxmem=*number*	“-qmaxmem” on page 229
[no]mbcs	“-qmbcs, -qdbcs” on page 230
[no]optimize=*number*	“-O, -qoptimize” on page 236
proclocal, procimported, procunknown	“-qprocimported, -qproclocal, -qprocunknown” on page 260
[no]proto	“-qproto” on page 262
[no]ro	“-qro” on page 267
[no]roconst	“-qroconst” on page 268
[no]showinc	“-qshowinc” on page 275
[no]source	“-qsource” on page 286
spill=*number*	“-qspill” on page 289
[no]srcmsg	“-qsrcmsg” on page 290
[no]stdinc	“-qstdinc” on page 292
[no]strict	“-qstrict” on page 294
tbtable=*option*	“-qtbtable” on page 305
tune=*option*	“-qtune” on page 310

Valid settings for #pragma options *option_keyword*	Compiler option equivalent
[no]unroll=[*yes/no/auto/n*]	"-qunroll" on page 314
[no]upconv	"-qupconv" on page 318
[no]xref	"-qxref" on page 330

Usage

Most **#pragma options** directives must come before any statements in your source program; only comments, blank lines, and other pragma specifications can precede them. For example, the first few lines of your program can be a comment followed by the **#pragma options** directive:

```
/* The following is an example of a #pragma options directive: */

#pragma options langlvl=stdc89 halt=s spill=1024 source

/* The rest of the source follows ... */
```

To specify more than one compiler option with the **#pragma options** directive, separate the options using a blank space. For example:

```
#pragma options langlvl=stdc89 halt=s spill=1024 source
```

#pragma option_override

Category

Optimization and tuning

Purpose

Allows you to specify optimization options at the subprogram level that override optimization options given on the command line.

This enables finer control of program optimization, and can help debug errors that occur only under optimization.

Syntax

```
►►─#─pragma─option_override──────────────────────────────────────────────►

               ┌────────────────────────────────────────────────────┐
               ▼                                                    │
►─(─identifier───,───"─opt─(──┬─size──┬──────────┬──────────────┬─)─"───┴─)──────►◄
                              │       ├─, yes─┤               │
                              │       └─, no──┘               │
                              ├─level─,──┬─0─┬────────────────┤
                              │          ├─2─┤                │
                              │          └─3─┘                │
                              ├─registerspillsize─,──size─────┤
                              │        ┌─,─────────────────┐  │
                              │        ▼                   │  │
                              └─strict───┬─yes────────────┬┴──┘
                                         ├─no─────────────┤
                                         └─suboption_list─┘
```

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]
registerspillsize, *size*	-qspill=*size*
size	-qcompact
size, yes	
size, no	-qnocompact
strict	-qstrict, -qstrict=all
strict, yes	
strict, no	-qnostrict
strict, *suboption_list*	-qstrict=*suboption_list*

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.

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 236
- "-qcompact" on page 122
- "-qspill" on page 289
- "-qstrict" on page 294

#pragma pack

Category

Object code control

Purpose

Sets the alignment of all aggregate members to a specified byte boundary.

If the byte boundary number is smaller than the natural alignment of a member, padding bytes are removed, thereby reducing the overall structure or union size.

Syntax

```
►►──#──pragma──pack──(──┬─────────┬──)──────────────────────────────────────►◄
                        ├─nopack──┤
                        ├─number──┤
                        └─pop─────┘
```

Defaults

Members of aggregates (structures, unions, and classes) are aligned on their natural boundaries and a structure ends on its natural boundary. The alignment of an aggregate is that of its strictest member (the member with the largest alignment requirement).

Parameters

nopack
: Disables packing. A warning message is issued and the pragma is ignored.

number
: is one of the following:

1 Aligns structure members on 1-byte boundaries, or on their natural alignment boundary, whichever is less.

2 Aligns structure members on 2-byte boundaries, or on their natural alignment boundary, whichever is less.

4 Aligns structure members on 4-byte boundaries, or on their natural alignment boundary, whichever is less.

8 Aligns structure members on 8-byte boundaries, or on their natural alignment boundary, whichever is less.

16 Aligns structure members on 16-byte boundaries, or on their natural alignment boundary, whichever is less.

pop
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 aligns all bit fields in a structure/union on 1-bit boundaries. Example:

```
#pragma pack(2)
struct A{
  int a:31;
  int b:2;
}x;

int main(){
  printf("size of struct A = %lu\n", sizeof(x));
}
```

When the program is compiled and run, the output is:

```
size of struct A = 6
```

But if you remove the **#pragma pack** directive, you get this output:

```
size of struct A = 8
```

The **#pragma pack** directive applies only to complete declarations of structures or unions; this excludes forward declarations, in which member lists are not specified. For example, in the following code fragment, the alignment for `struct S` is 4, since this is the rule in effect when the member list is declared:

```
#pragma pack(1)
struct S;
#pragma pack(4)
struct S { int i, j, k; };
```

A nested structure has the alignment that precedes its declaration, not the alignment of the structure in which it is contained, as shown in the following example:

```
#pragma pack (4)                // 4-byte alignment
         struct nested {
           int  x;
           char y;
           int  z;
         };

         #pragma pack(1)           // 1-byte alignment
         struct packedcxx{
            char   a;
            short  b;
            struct nested  s1;     // 4-byte alignment
         };
```

If more than one **#pragma pack** directive appears in a structure defined in an inlined function, the **#pragma pack** directive in effect at the beginning of the structure takes precedence.

Examples

The following example shows how the **#pragma pack** directive can be used to set the alignment of a structure definition:

```
//  header file file.h

   #pragma pack(1)

   struct jeff{           //    this structure is packed
     short bill;          //    along 1-byte boundaries
     int *chris;
   };
   #pragma pack(pop)       //   reset to previous alignment rule
// source file anyfile.c

   #include "file.h"

   struct jeff j;           //   uses the alignment specified
                            //   by the pragma pack directive
                            //   in the header file and is
                            //   packed along 1-byte boundaries
```

This example shows how a **#pragma pack** directive can affect the size and mapping of a structure:

```
struct s_t {
 char a;
 int b;
 short c;
 int d;
}S;
```

Default mapping:	**With #pragma pack(1):**
size of s_t = 16	size of s_t = 11
offset of a = 0	offset of a = 0
offset of b = 4	offset of b = 1
offset of c = 8	offset of c = 5
offset of d = 12	offset of d = 7
alignment of a = 1	alignment of a = 1
alignment of b = 4	alignment of b = 1
alignment of c = 2	alignment of c = 1

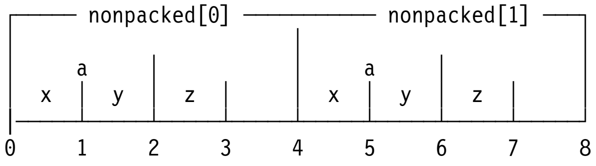
Default mapping:	**With #pragma pack(1):**
alignment of d = 4	alignment of d = 1

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:



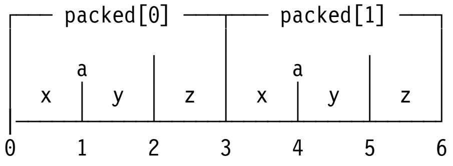
The next example uses **#pragma pack(1)** to set the alignment of unions of type uu to 1 byte:

```
#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

- “-qalign” on page 98
- "Using alignment modifiers" in the *XL C Optimization and Programming Guide*

#pragma reachable

Category

Optimization and tuning

Purpose

Informs the compiler that the point in the program after a named function can be the target of a branch from some unknown location.

By informing the compiler that the instruction after the specified function can be reached from a point in your program other than the return statement in the named function, the pragma allows for additional opportunities for optimization.

Note: The compiler automatically inserts **#pragma reachable** directives for the `setjmp` family of functions (`setjmp`, `_setjmp`, `sigsetjmp`, and `_sigsetjmp`) when you include the setjmp.h header file.

Syntax

```
                          ┌─,───────────┐
                          ▼             │
►►─#─pragma─reachable─(───function_name─┴─)──────────────────────►◄
```

Parameters

function_name
The name of a function preceding the instruction which is reachable from a point in the program other than the function's return statement.

Defaults

Not applicable.

Related information

- "#pragma leaves" on page 356

#pragma reg_killed_by

Category

Optimization and tuning

Purpose

Specifies registers that may be altered by functions specified by **#pragma mc_func**.

Ordinarily, code generated for functions specified by **#pragma mc_func** may alter any or all volatile registers available on your system. You can use **#pragma reg_killed_by** to explicitly list a specific set of volatile registers to be altered by such functions. Registers not in this list will not be altered.

Syntax

```
                                    ┌─,─────────────────────────┐
                                    ▼                           │
►►─#─pragma─reg_killed_by─function───┬───────────────────────┬─┴────►◄
                                     └─register─┬──────────┬─┘
                                                └─-─register┘
```

Parameters

function
The name of a function previously defined using the **#pragma mc_func** directive.

register
The symbolic name(s) of either a single register or a range of registers to be altered by the named *function*. The symbolic name must be a valid register name on the target platform. Valid registers are:

cr0, cr1, and cr5 to cr7
Condition registers

ctr Count register

gr0 and gr3 to gr12
General purpose registers

fp0 to fp13
Floating-point registers

fsr Floating-point and status control register

lr Link register

vr0 to vr31
Vector registers (on selected processors only)

xer Fixed-point exception register

You can identify a range of registers by providing the symbolic names of both starting and ending registers, separated by a dash.

If no *register* is specified, no volatile registers will be killed by the named *function*.

Examples

The following example shows how to use **#pragma reg_killed_by** to list a specific set of volatile registers to be used by the function defined by **#pragma mc_func**.

```
int add_logical(int, int);
#pragma mc_func add_logical {"7c632014" "7c630194"}
                /*   addc       r3 <- r3, r4          */
                /*   addze      r3 <- r3, carry bit    */

#pragma reg_killed_by add_logical gr3, xer
                /* only gpr3 and the xer are altered by this function */


main() {

      int i,j,k;

      i = 4;
      k = -4;
      j = add_logical(i,k);
      printf("\n\nresult = %d\n\n",j);
}
```

Related information

- "#pragma mc_func" on page 359

#pragma simd_level

Category

Optimization and tuning

Purpose

Controls the compiler code generation of vector instructions for individual loops.

Vector instructions can offer high performance when used with algorithmic-intensive tasks such as multimedia applications. You have the flexibility to control the aggressiveness of autosimdization on a loop-by-loop basis, and might be able to achieve further performance gain with this fine grain control.

The supported levels are from 0 to 10. level(0) indicates performing no autosimdization on the loop that follows the pragma directive. level(10) indicates performing the most aggressive form of autosimdization on the loop. With this pragma directive, you can control the autosimdization behavior on a loop-by-loop basis.

Syntax

```
►►─#─pragma─simd_level─(─n─)──────────────────────────────────────►◄
```

Parameters

n A scalar integer initialization expression, from 0 to 10, specifying the aggressiveness of autosimdization on the loop that follows the pragma directive.

Usage

A loop with no simd_level pragma is set to simd level 5 by default, if **-qsimd=auto** is in effect.

#pragma simd_level(0) is equivalent to **#pragma nosimd**, where autosimdization is not performed on the loop that follows the pragma directive.

#pragma simd_level(10) instructs the compiler to perform autosimdization on the loop that follows the pragma directive most aggressively, including bypassing cost analysis.

Rules

The rules of **#pragma simd_level** directive are listed as follows:

- The **#pragma simd_level** directive has effect only for architectures that support vector instructions and when used with **-qsimd=auto**.
- The **#pragma simd_level** directive applies to `while`, `do while`, and `for` loops.
- 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 169 in this document and "Using OpenMP directives" in the *IBM XL C Optimization and Programming Guide*.

Examples

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

} ...
```

#pragma STDC CX_LIMITED_RANGE

Category

Optimization and tuning

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

```
                                         ┌─off─────┐
►►──#──pragma──STDC cx_limited_range─────┼─on──────┼──────────────────────────────────►◄
                                         └─default─┘
```

Usage

Using values outside the limited range may generate wrong results, where the limited range is defined such that the "obvious symbolic definition" will not overflow or run out of precision.

The pragma is effective from its first occurrence until another **cx_limited_range** pragma is encountered, or until the end of the translation unit. When the pragma occurs inside a compound statement (including within a nested compound statement), it is effective from its first occurrence until another **cx_limited_range** pragma is encountered, or until the end of the compound statement.

Examples

The following example shows the use of the pragma for complex division:

```
#include <complex.h>

_Complex double a, b, c, d;
void p() {

d = b/c;

{

#pragma STDC CX_LIMITED_RANGE ON
a = b / c;

}
}
```

The following example shows the use of the pragma for complex absolute value:

```
#include <complex.h>

_Complex double cd = 10.10 + 10.10*I;
int p() {

#pragma STDC CX_LIMITED_RANGE ON

double d = cabs(cd);
}
```

Related information

- "Standard pragmas" in the *XL C Language Reference*

#pragma stream_unroll

Category

Optimization and tuning

Purpose

When optimization is enabled, breaks a stream contained in a `for` loop into multiple streams.

Syntax

```
►►──#──pragma──stream_unroll──┬──────────────┬──────────────────────────────►◄
                              └─(──number──)─┘
```

Parameters

number
: A loop unrolling factor. The value of *number* is a positive integral constant expression.

  An unroll factor of 1 disables unrolling.

If *number* is not specified, the optimizer determines an appropriate unrolling factor for each nested loop.

Usage

To enable stream unrolling, you must specify **-qhot** and **-qstrict**, or **-qsmp**, or use optimization level **-O4** or higher. If **-qstrict** is in effect, no stream unrolling takes place.

For stream unrolling to occur, the **#pragma stream_unroll** directive must be the last pragma specified preceding a `for` loop. Specifying **#pragma stream_unroll** more than once for the same `for` loop or combining it with other loop unrolling pragmas (**#pragma unroll**, **#pragma nounroll**, **#pragma unrollandfuse**, **#pragma nounrollandfuse**) results in a warning.

Examples

The following example shows how **#pragma stream_unroll** can increase performance.

```
int i, m, n;
int a[1000];
int b[1000];
int c[1000];

....

#pragma stream_unroll(4)
for (i=0; i<n; i++) {
  a[i] = b[i] * c[i];
}
```

The unroll factor of 4 reduces the number of iterations from `n` to `n/4`, as follows:

```
m = n/4;

for (i=0; i<n/4; i++){
  a[i] = b[i] + c[i];
  a[i+m] = b[i+m] + c[i+m];
  a[i+2*m] = b[i+2*m] + c[i+2*m];
  a[i+3*m] = b[i+3*m] + c[i+3*m];
}
```

The increased number of read and store operations are distributed among a number of streams determined by the compiler, which reduces computation time and increase performance.

Related information

- “-qunroll” on page 314
- “#pragma unrollandfuse”

#pragma strings

See “-qro” on page 267.

#pragma unroll, #pragma nounroll

See “-qunroll” on page 314

#pragma unrollandfuse

Category

Optimization and tuning

Purpose

Instructs the compiler to attempt an unroll and fuse operation on nested `for` loops.

Syntax

```
►►─#─pragma─┬─nounrollandfuse──────────────────┬──────────────────────────►◄
            └─unrollandfuse─┬────────────────┬─┘
                            └─(─number─)─────┘
```

Parameters

number
: A loop unrolling factor. The value of *number* is a positive integral constant expression.

If *number* is not specified, the optimizer determines an appropriate unrolling factor for each nested loop.

Usage

The **#pragma unrollandfuse** directive applies only to the outer loops of nested `for` loops that meet the following conditions:

- There must be only one loop counter variable, one increment point for that variable, and one termination variable. These cannot be altered at any point in the loop nest.
- Loops cannot have multiple entry and exit points. The loop termination must be the only means to exit the loop.
- Dependencies in the loop must not be "backwards-looking". For example, a statement such as `A[i][j] = A`**`[i -1]`**`[j + 1] + 4)` must not appear within the loop.

For loop unrolling to occur, the **#pragma unrollandfuse** directive must precede a `for` loop. You must not specify **#pragma unrollandfuse** for the innermost `for` loop.

You must not specify **#pragma unrollandfuse** more than once, or combine the directive with **#pragma nounrollandfuse**, **#pragma nounroll**, **#pragma unroll**, or **#pragma stream_unroll** directives for the same `for` loop.

Predefined macros

None.

Examples

In the following example, a **#pragma unrollandfuse** directive replicates and fuses the body of the loop. This reduces the number of cache misses for array `b`.

```
int i, j;
int a[1000][1000];
int b[1000][1000];
int c[1000][1000];


....

#pragma unrollandfuse(2)
for (i=1; i<1000; i++) {
    for (j=1; j<1000; j++) {
        a[j][i] = b[i][j] * c[j][i];
    }
}
```

The `for` loop below shows a possible result of applying the **#pragma unrollandfuse(2)** directive to the loop shown above:

```
for (i=1; i<1000; i=i+2) {
    for (j=1; j<1000; j++) {
        a[j][i] = b[i][j] * c[j][i];
        a[j][i+1] = b[i+1][j] * c[j][i+1];
    }
}
```

You can also specify multiple **#pragma unrollandfuse** directives in a nested loop structure.

```
int i, j, k;
int a[1000][1000];
int b[1000][1000];
int c[1000][1000];
int d[1000][1000];
int e[1000][1000];


....

#pragma unrollandfuse(4)
for (i=1; i<1000; i++) {
#pragma unrollandfuse(2)
    for (j=1; j<1000; j++) {
   for (k=1; k<1000; k++) {
            a[j][i] = b[i][j] * c[j][i] + d[j][k] * e[i][k];
        }
    }
}
```

Related information

- "-qunroll" on page 314
- "#pragma stream_unroll" on page 374

#pragma weak

Category

Object code control

Purpose

Prevents the linker from issuing error messages if it encounters a symbol multiply-defined during linking, or if it does not find a definition for a symbol.

The pragma can be used to allow a program to call a user-defined function that has the same name as a library function. By marking the library function definition as "weak", the programmer can reference a "strong" version of the function and cause the linker to accept multiple definitions of a global symbol in the object code. While this pragma is intended for use primarily with functions, it will also work for most data objects.

Syntax

```
►►─#─pragma─weak─name1─┬───────────┬──────────────────────────────►◄
                       └─=─name2───┘
```

Parameters

name1
    A name of a data object or function with external linkage.

name2
A name of a data object or function with external linkage.

name2 must not be a member function. If *name2* is a template function, you must explicitly instantiate the template function.

Usage

There are two forms of the **weak** pragma:

#pragma weak *name1*
This form of the pragma marks the definition of the *name1* as "weak" in a given compilation unit. If *name1* is referenced from anywhere in the program, the linker will use the "strong" version of the definition (that is, the definition not marked with **#pragma weak**), if there is one. If there is no strong definition, the linker will use the weak definition; if there are multiple weak definitions, it is unspecified which weak definition the linker will select (typically, it uses the definition found in the first object file specified on the command line during the link step). *name1* must be defined in the same compilation unit as **#pragma weak**.

#pragma weak *name1*=*name2*
This form of the pragma creates a weak definition of the *name1* for a given compilation unit, and an alias for *name2*. If *name1* is referenced from anywhere in the program, the linker will use the "strong" version of the definition (that is, the definition not marked with **#pragma weak**), if there is one. If there is no strong definition, the linker will use the weak definition, which resolves to the definition of *name2*. If there are multiple weak definitions, it is unspecified which weak definition the linker will select (typically, it uses the definition found in the first object file specified on the command line during the link step).

name2 must be defined in the same compilation unit as **#pragma weak**. *name1* may or may not be declared in the same compilation unit as the **#pragma weak**, but must never be defined in the compilation unit. If *name1* is declared in the compilation unit, *name1*'s declaration must be compatible to that of *name2*. For example, if *name2* is a function, *name1* must have the same return and argument types as *name2*.

This pragma should not be used with uninitialized global data, or with shared library data objects that are exported to executables.

Examples

The following is an example of the **#pragma weak** *name1* form:

```
// Compilation unit 1:

#include <stdio.h>

void foo();

int main()
{
      foo();
}

// Compilation unit 2:

#include <stdio.h>
```

```
#pragma weak foo

void foo()
{
       printf("Foo called from compilation unit 2\n");
}

// Compilation unit 3:

#include <stdio.h>

void foo()
{
       printf("Foo called from compilation unit 3\n");
}
```

If all three compilation units are compiled and linked together, the linker will use the strong definition of `foo` in compilation unit 3 for the call to `foo` in compilation unit 1, and the output will be:

```
Foo called from compilation unit 3
```

If only compilation unit 1 and 2 are compiled and linked together, the linker will use the weak definition of `foo` in compilation unit 2, and the output will be:

```
Foo called from compilation unit 2
```

The following is an example of the **#pragma weak** *name1*=*name2* form:

```
// Compilation unit 1:

#include <stdio.h>

void foo();

int main()
{
foo();
}

// Compilation unit 2:

#include <stdio.h>

void foo(); // optional


#pragma weak foo = foo2

void foo2()
{
printf("Hello from foo2!\n");
}

// Compilation unit 3:

#include <stdio.h>

void foo()
{
printf("Hello from foo!\n");
}
```

If all three compilation units are compiled and linked together, the linker will use the strong definition of `foo` in compilation unit 3 for the call to `foo` from compilation unit 1, and the output will be:

```
Hello from foo!
```

If only compilation unit 1 and 2 are compiled and linked together, the linker will use the weak definition of `foo` in compilation unit 2, which is an alias for `foo2`, and the output will be:

```
Hello from foo2!
```

Related information

- "The weak variable attribute" in the *XL C Language Reference*
- "The weak function attribute" in the *XL C Language Reference*
- “#pragma map” on page 358
- “-qweaksymbol” on page 329
- “-qweakexp” on page 328

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.

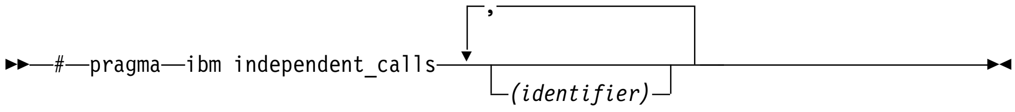
You can use IBM SMP or OpenMP directives in C programs. Each has its own usage characteristics.

#pragma ibm independent_calls

Description

The **independent_calls** pragma asserts that specified function calls within the chosen loop have no loop-carried dependencies. This information helps the compiler perform dependency analysis.

Syntax



Where *identifier* is a comma-separated list that represents the name of the functions.

Usage

identifier cannot be the name of a pointer to a function.

If no function identifiers are specified, the compiler assumes that all functions inside the loop are free of carried dependencies.

#pragma ibm permutation

Purpose

The **permutation** pragma asserts that on the following loop, different elements of the named arrays are guaranteed to have different values (that is, `a[i] == a[j] iff i == j`).

Syntax

```
                                   ┌─,──────────┐
                                   ▼            │
►►──#──pragma──ibm permutation───(identifier)─┴──────────────────────────►◄
```

where *identifier* represents the name of an array. The *identifier* cannot be a function parameter or the name of a pointer.

Usage

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

This assertion may enable loop transformations if elements are used to index other arrays. This pragma is useful for programs that deal with sparse data structures.

#pragma ibm schedule

Purpose

Note: #pragma ibm schedule has been deprecated and might be removed in a future release. Use "#pragma omp parallel for" on page 393 with the **schedule** clause. For more information about SMP directives, see "Deprecated directives" on page 339.

The **schedule** pragma specifies the scheduling algorithms used for parallel processing.

Syntax

```
►►──#──pragma──ibm schedule──(sched-type)────────────────────────────►◄
```

Parameters

sched-type represents one of the following options:

`affinity`

Iterations of a loop are initially divided into local partitions of size **ceiling**(*number_of_iterations*/*number_of_threads*) contiguous iterations. Each local partition is then further subdivided into chunks of size **ceiling**(*number_of_iterations_remaining_in_partition*/2).

When a thread becomes available, it takes the next chunk from its local partition. If there are no more chunks in the local partition, the thread takes an available chunk from the partition of another thread.

affinity,*n*
As above, except that each local partition is subdivided into chunks of size *n* contiguous iterations. *n* must be an integral assignment expression of value 1 or greater.

dynamic
Iterations of a loop are divided into chunks, each of which contains one iteration

Chunks are assigned to threads on a first-come, first-do basis as threads become available. This continues until all work is completed.

dynamic,*n*
Iterations of a loop are divided into chunks that contain *n* contiguous iterations each. The final chunk might contain fewer than *n* iterations.

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

guided
Chunks are made progressively smaller until a chunk size of one is reached. The first chunk is of size **ceiling**(*number_of_iterations/number_of_threads*) contiguous iterations. Remaining chunks are of size **ceiling**(*number_of_iterations_remaining/number_of_threads*).

Chunks are assigned to threads on a first-come, first-serve basis as threads become available. This continues until all work is completed.

guided,*n*
As above, except the minimum chunk size for all the chunks but the last chunk is set to *n* contiguous iterations. *n* must be an integral assignment expression of value 1 or greater.

runtime
Scheduling policy is determined at run time.

static
Iterations of a loop are divided into chunks of size of at least **floor**(*number_of_iterations/number_of_threads*) contiguous iterations. The first **remainder**(*number_of_iterations/number_of_threads*) chunks have one more iteration. 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* contiguous iterations except for the last iteration. Each chunk is assigned to a thread in *round-robin* fashion.

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

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*

Usage

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

Scheduling algorithms for parallel processing can be specified using any of the methods shown below. If used, methods higher in the list override entries lower in the list.

- pragma statements
- compiler command line options
- runtime command line options
- runtime default options

Scheduling algorithms can also be specified using the **schedule** argument of the **independent_loop** pragma statement. If different scheduling types are specified for a given loop, the last one specified is applied.

#pragma ibm sequential_loop

Purpose

The **sequential_loop** pragma explicitly instructs the compiler to execute the chosen loop sequentially.

Syntax

```
►►──#──pragma──ibm sequential_loop─────────────────────────────────────►◄
```

Usage

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

This pragma disables automatic parallelization of the chosen loop, and is always respected by the compiler.

#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

```
                         ┌─update──┐
►►──#──pragma──omp atomic─┼─────────┼──────────────────────────────────►◄
                         ├─read────┤
                         ├─write───┤
                         └─capture─┘
```

```
►►──expression_statement───────────────────────────────────────────────►◄
```

Syntax form 2

```
►►—#—pragma—omp atomic—capture——————————————————————————————————————►◄
```

```
►►—structured_block—————————————————————————————————————————————————►◄
```

where *expression_statement* is an expression statement of scalar type, and *structured_block* is a structured block of two expression statements.

Clauses

update

Updates the value of a variable atomically. Guarantees that only one thread at a time updates the shared variable, avoiding errors from simultaneous writes to the same variable. An **omp atomic** directive without a clause is equivalent to an **omp atomic** update.

Note: Atomic updates cannot write arbitrary data to the memory location, but depend on the previous data at the memory location.

read

Reads the value of a variable atomically. The value of a shared variable can be read safely, avoiding the danger of reading an intermediate value of the variable when it is accessed simultaneously by a concurrent thread.

write

Writes the value of a variable atomically. The value of a shared variable can be written exclusively to avoid errors from simultaneous writes.

capture

Updates the value of a variable while capturing the original or final value of the variable atomically.

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;`	

Directive clause	*expression_statement*	*structured_block*

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

```
                                 ┌─,────────┐
                                 ▼          │
►►──#──pragma──omp parallel────────clause───┴──────────────────────────────►◄
```

Parameters

clause is any of the following clauses:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Usage

When a parallel region is encountered, a logical team of threads is formed. Each thread in the team executes all statements within a parallel region except for work-sharing constructs. Work within work-sharing constructs is distributed among the threads in a team.

Loop iterations must be independent before the loop can be parallelized. An implied barrier exists at the end of a parallelized statement block.

By default, nested parallel regions are serialized.

Related information:

"OMP_NESTED" on page 33

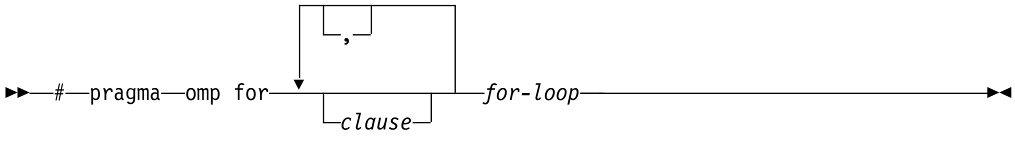
"OMP_PROC_BIND" on page 35

#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.

```
►►—COLLAPSE—(—n—)——————————————————————————————————►◄
```

- Only one collapse clause is allowed on a worksharing **for** or **parallel for** pragma.
- The specified number of loops must be present lexically. That is, none of the loops can be in a called subroutine.
- The loops must form a rectangular iteration space and the bounds and stride of each loop must be invariant over all the loops.
- If the loop indices are of different size, the index with the largest size will be used for the collapsed loop.
- The loops must be perfectly nested; that is, there is no intervening code nor any OpenMP pragma between the loops which are collapsed.
- The associated do-loops must be structured blocks. Their execution must not be terminated by an **break** statement.
- If multiple loops are associated to the loop construct, only an iteration of the innermost associated loop may be curtailed by a **continue** statement. If multiple loops are associated to the loop construct, there must be no branches to any of the loop termination statements except for the innermost associated loop.

Ordered construct

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

Lastprivate clause

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

Other SMP and performance pragmas

stream_unroll,**unroll**,**unrollandfuse**,**nounrollandfuse** pragmas cannot be used for any of the loops associated with the **collapse** clause loop nest. The **independent_loop** pragma can be used for any of the loops associated with the **collapse** clause. **independent_loop** is not OpenMP specific.

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`:
  - `_Bool`
  - `char`
  - `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

```
►►—#—pragma—omp ordered————————————————————————————————————————————►◄
```

Usage

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

- It must appear within the extent of a **omp for** or **omp parallel for** construct containing an **ordered** clause.
- It applies to the statement block immediately following it. Statements in that block are executed in the same order in which iterations are executed in a sequential loop.

- An iteration of a loop must not execute the same **omp ordered** directive more than once.
- An iteration of a loop must not execute more than one distinct **omp ordered** directive.

#pragma omp parallel for

Purpose

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

Syntax

```
                                  ┌──────────────┐
                                  │ └─,─┘        │
                                  ▼              │
►►──#──pragma──omp parallel for─────────────────┴──for-loop──────────────────►◄
                                    └─clause─┘
```

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

```
                                   ┌─,────┐
                                   ▼      │
►►──#──pragma──omp sections────clause──┴──────────────────────────────────►◄
```

Parameters

clause is any of the following clauses:

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

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

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

reduction (*operator*: *list*)

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

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

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

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

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

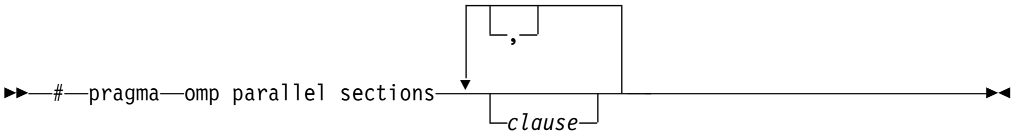
When program execution reaches a **omp sections** directive, program segments defined by the following **omp section** directive are distributed for parallel execution among available threads. A barrier is implicitly defined at the end of the larger program region associated with the **omp sections**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

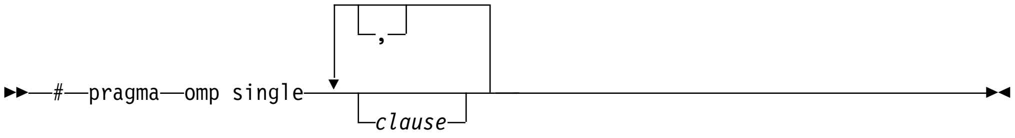
All clauses and restrictions described in the **omp parallel** and **omp sections** directives apply to the **omp parallel sections** directive.

#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

```
►►──#──pragma──omp master──────────────────────────────────────────►◄
```

Usage

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

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

#pragma omp critical

Purpose

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

Syntax

```
                              ┌─,──────┐
                              ▼        │
►►──#──pragma──omp critical─────(name)─┴───────────────────────────►◄
```

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

Usage

A thread waits at the start of a critical region identified by a given name until no other thread in the program is executing a critical region with that same name.

Critical sections not specifically named by **omp critical** directive invocation are mapped to the same unspecified name.

#pragma omp barrier

Purpose

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

Syntax

```
►►──#──pragma──omp barrier─────────────────────────────────────────────►◄
```

Usage

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

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

#pragma omp 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

```
                                ┌──────────┐
                                │ ┌─,──┐   │
                                ▼ │    │   │
►►──#──pragma──omp flush────────┴─┴list┴───┴──────────────────────────►◄
```

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:

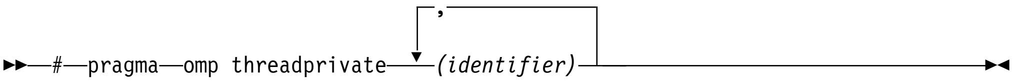
```
if (x!=0) {
   #pragma omp flush    /* valid usage    */
}
if (x!=0)
   #pragma omp flush    /* invalid usage  */
```

#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

```
                              ┌─,──────┐
                              ▼        │
►►──#──pragma──omp task─────clause─┴──────────────────────────────────────►◄
```

Parameters

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

`default (shared | none)`

Defines the default data scope of variable in each task. Only one `default` clause can be specified on an `omp task` directive.

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

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

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

`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 401

#pragma omp taskyield

Purpose

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

Syntax

```
►►──#──pragma──omp taskyield──────────────────────────────────────────►◄
```

#pragma omp taskwait

Purpose

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

Syntax

```
►►──#──pragma──omp taskwait───────────────────────────────────────►◄
```

Related reference:

"#pragma omp task" on page 399

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 405
- “Macros related to compiler features” on page 405

“Examples of predefined macros” on page 412 show how you can use them in your code.

General macros

The following predefined macros are always predefined by the compiler. Unless noted otherwise, all the following macros are *protected*, which means that the compiler will issue a warning if you try to undefine or redefine them.

Table 34. General predefined macros

Predefined macro name	**Description**	**Predefined value**
__BASE_FILE__	Indicates the name of the primary source file.	The fully qualified file name of the primary source file.

__COUNTER__	Expands to an integer that starts from 0. The value increases by 1 each time this macro is expanded.
	You can use this macro with the ## operator to generate unique variable or function names. The following example shows the declaration of distinct identifiers with a single token:
	`#define CONCAT(a, b) a##b`
	`#define CONCAT_VAR(a, b) CONCAT(a, b)`
	`#define VAR CONCAT_VAR(var, __COUNTER__)`
	
	`//Equivalent to int var0 = 1;`
	`int VAR = 1;`
	
	`//Equivalent to char var1 = 'a';`
	`char VAR = 'a';`
		An integer variable that starts from 0. The value increases by 1 each time this macro is expanded.

__DATE__	Indicates the date that the source file was preprocessed.	A character string containing the date when the source file was preprocessed.
__FILE__	Indicates the name of the preprocessed source file.	A character string containing the name of the preprocessed source file.
__FUNCTION__	Indicates the name of the function currently being compiled.	A character string containing the name of the function currently being compiled.
__LINE__	Indicates the current line number in the source file.	An integer constant containing the line number in the source file.

Table 34. General predefined macros (continued)

Predefined macro name	Description	Predefined value
__SIZE_TYPE__	Indicates the underlying type of `size_t` on the current platform. Not protected.	`unsigned int` in 32-bit compilation mode and `unsigned long` in 64-bit compilation mode.
__TIME__	Indicates the time that the source file was preprocessed.	A character string containing the time when the source file was preprocessed.
__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 compiler

Macros related to the XL 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 **-qshowmacros=pre -E** compiler options to view the values of the predefined macros.

Table 35. Compiler product predefined macros

Predefined macro name	Description	Predefined value
__IBMC__	Indicates the level of the XL C compiler.	An integer in format *VRM*, where: *V* Represents the version number *R* Represents the release number *M* Represents the modification number
__xlc__	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

Table 35. Compiler product predefined macros (continued)

Predefined macro name	Description	Predefined value
__xlC__	Indicates the VR level of the XL C compilers in hexadecimal format.	A 4-digit hexadecimal integer in format 0x*VVRR*, where: *V* Represents the version number *R* Represents the release number
__xlC_ver__	Indicates the MF level of the XL C compilers in hexadecimal format.	An 8-digit hexadecimal integer in format 0x*0000MMFF*, where: *M* Represents the modification number *F* Represents the fix level For example, in PTF 3, the value of the macro is 0x00000003.

Macros related to the platform

The following predefined macros are provided to facilitate porting applications between platforms. All platform-related predefined macros are unprotected and can be undefined or redefined without warning unless otherwise specified.

Table 36. Platform-related predefined macros

Predefined macro name	Description	Predefined value	Predefined under the following conditions
_BIG_ENDIAN, __BIG_ENDIAN__	Indicates that the platform is big-endian (that is, the most significant byte is stored at the memory location with the lowest address).	1	Always predefined.
__powerpc, __powerpc__	Indicates that the target is a Power architecture.	1	Predefined when the target is a Power architecture.
__PPC, __PPC__	Indicates that the target is a Power architecture.	1	Predefined when the target is a Power architecture.
__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” on page 406
- “Macros related to architecture settings” on page 407
- “Macros related to language levels” on page 408

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 37. General option-related predefined macros

Predefined macro name	Description	Predefined value	Predefined when the following compiler option or equivalent pragma is in effect
__64BIT__	Indicates that 64-bit compilation mode is in effect.	1	-q64
__ALTIVEC__	Indicates support for vector data types. (unprotected)	1	-qaltivec
_CHAR_SIGNED, __CHAR_SIGNED__	Indicates that the default character type is `signed char`.	1	-qchars=signed
_CHAR_UNSIGNED, __CHAR_UNSIGNED__	Indicates that the default character type is `unsigned char`.	1	-qchars=unsigned
__DEBUG_ALLOC__	Indicates that debug versions of the standard memory management functions are being used.	1	-qheapdebug
__IBM_GCC_ASM	Indicates support for GCC inline `asm` statements.	1	**-qasm=gcc** and **-qlanglvl=extc99** | **extc89** | **extended** or **-qkeyword=asm**
		0	**-qnoasm** and **-qlanglvl=extc99** | **extc89** | **extended** or **-qkeyword=asm**
__IBM_DFP__	Indicates support for decimal floating-point types.	1	-qdfp
__IBM_DFP_SW_EMULATION__	Indicates that decimal floating-point computations are implemented through software emulation rather than in hardware instructions.	1	-qfloat=dfpemulate
_IBMSMP	Indicates that IBM SMP directives are recognized.	1	-qsmp

Table 37. General option-related predefined macros (continued)

Predefined macro name	Description	Predefined value	Predefined when the following compiler option or equivalent pragma is in effect
__IBM_UTF_LITERAL	Indicates support for UTF-16 and UTF-32 string literals.	1	-qlanglvl=extended
__LONGDOUBLE64	Indicates that the size of a `long double` type is 64 bits.	1	**-qnoldbl128**
__LONGDOUBLE128	Indicates that the size of a `long double` type is 128 bits.	1	-qldbl128
__OPTIMIZE__	Indicates the level of optimization in effect.	2	-O | -O2
		3	-O3 | -O4 | -O5
__OPTIMIZE_SIZE__	Indicates that optimization for code size is in effect.	1	-O | -O2 | -O3 | -O4 | -O5 and -qcompact
__VEC__	Indicates support for vector data types.	10206	-qaltivec

Macros related to architecture settings

The following macros can be tested for target architecture settings. All of these macros are predefined to a value of 1 by a **-qarch** compiler option setting, or any other compiler option that implies that setting. If the **-qarch** suboption enabling the feature is not in effect, then the macro is undefined.

Table 38. **-qarch***-related macros*

Macro name	Description	Predefined by the following -qarch suboptions
_ARCH_PPC	Indicates that the application is targeted to run on any Power processor.	Defined for all -qarch suboptions except auto.
_ARCH_PPC64	Indicates that the application is targeted to run on Power processors with 64-bit support.	ppc64 | ppc64gr | ppc64grsq | ppc64v | pwr4 | pwr5 | pwr5x | pwr6 | pwr6e | pwr7 | pwr8 | ppc970
_ARCH_PPCGR	Indicates that the application is targeted to run on Power processors with graphics support.	ppcgr | ppc64gr | ppc64grsq | ppc64v | pwr4 | pwr5 | pwr5x | pwr6 | pwr6e | pwr7 | pwr8 | ppc970
_ARCH_PPC64GR	Indicates that the application is targeted to run on Power processors with 64-bit and graphics support.	ppc64gr | ppc64v | pwr4 | pwr5 | pwr5x | pwr6 | pwr6e | pwr7 | pwr8 | ppc970
_ARCH_PPC64GRSQ	Indicates that the application is targeted to run on Power processors with 64-bit, graphics, and square root support.	ppc64grsq | ppc64v | pwr4 | pwr5 | pwr5x | pwr6 | pwr6e | pwr7 | pwr8 | ppc970
_ARCH_PPC64V	Indicates that the application is targeted to run on Power processors with 64-bit and vector processing support.	ppc64v | ppc970 | pwr6 | pwr6e | pwr7 | pwr8

Table 38. **-qarch**-related macros (continued)

Macro name	Description	Predefined by the following -qarch suboptions
_ARCH_PPC970	Indicates that the application is targeted to run on the PowerPC 970 processor.	ppc970
_ARCH_PWR4	Indicates that the application is targeted to run on POWER4 or higher processors.	pwr4 | pwr5 | pwr5x | pwr6 | pwr6e | pwr7 | pwr8 | ppc970
_ARCH_PWR5	Indicates that the application is targeted to run on POWER5 or higher processors.	pwr5 | pwr5x | pwr6 | pwr6e | pwr7 | pwr8
_ARCH_PWR5X	Indicates that the application is targeted to run on POWER5+ or higher processors.	pwr5x | pwr6 | pwr6e | pwr7 | pwr8
_ARCH_PWR6	Indicates that the application is targeted to run on POWER6 or higher processors.	pwr6 | pwr6e | pwr7 | pwr8
_ARCH_PWR6E	Indicates that the application is targeted to run on POWER6 processors running in POWER6 raw mode.	pwr6e
_ARCH_PWR7	Indicates that the application is targeted to run on POWER7 , POWER7+ or higher processors.	pwr7 | pwr8
_ARCH_PWR8	Indicates that the application is targeted to run on POWER8 processors.	pwr8

Related information

- "-qarch" on page 102

Macros related to language levels

The following macros can be tested for C99 features, features related to GNU C, and other IBM language extensions. All of these macros except __STDC_VERSION__ are predefined to a value of 1 by a specific language level, represented by a suboption of the **-qlanglvl** compiler option, or any invocation or pragma that implies that suboption. If the suboption enabling the feature is not in effect, then the macro is undefined. For descriptions of the features related to these macros, see the *XL C Language Reference*.

Table 39. Predefined macros for language features

Predefined macro name	Description	Predefined when the following language level is in effect
__C99_BOOL	Indicates support for the `_Bool` data type.	extc1x | stdc99 | extc99 | extc89 | extended
__C99_COMPLEX	Indicates that the support for C99 complex types is enabled or that the C99 complex header should be included.	extc1x | stdc99 | extc99 | extc89 | extended
__C99_COMPLEX_HEADER__	Indicates support for C99-style complex headers.	c99complexheader
__C99_CPLUSCMT	Indicates support for C++ style comments	extc1x | stdc99 | extc99 | stdc89 | extc89 | extended (also **-qcpluscmt**)

Table 39. Predefined macros for language features (continued)

Predefined macro name	Description	Predefined when the following language level is in effect
__C99_COMPOUND_LITERAL	Indicates support for compound literals.	extc1x | stdc99 | extc99 | extc89 | extended
__C99_DESIGNATED_INITIALIZER	Indicates support for designated initialization.	extc1x | stdc99 | extc99 | extc89 | extended
__C99_DUP_TYPE_QUALIFIER	Indicates support for duplicated type qualifiers.	extc1x | stdc99 | extc99 | extc89 | extended
__C99_EMPTY_MACRO_ARGUMENTS	Indicates support for empty macro arguments.	extc1x | stdc99 | extc99 | extc89 | extended
__C99_FLEXIBLE_ARRAY_MEMBER	Indicates support for flexible array members.	extc1x | stdc99 | extc99 | extc89 | extended
__C99__FUNC__	Indicates support for the `__func__` predefined identifier.	extc1x | stdc99 | extc99 | extc89 | extended
__C99_HEX_FLOAT_CONST	Indicates support for hexadecimal floating constants.	extc1x | stdc99 | extc99 | extc89 | extended
__C99_INLINE	Indicates support for the `inline` function specifier.	extc1x | stdc99 | extc99 (also **-qkeyword=inline**)
__C99_LLONG	Indicates support for C99-style `long long` data types and literals.	extc1x | stdc99 | extc99
__C99_MACRO_WITH_VA_ARGS	Indicates support for function-like macros with variable arguments.	extc1x | stdc99 | extc99 | extc89 | extended
__C99_MAX_LINE_NUMBER	Indicates that the maximum line number is 2147483647.	extc1x | stdc99 | extc99 | extc89 | extended
__C99_MIXED_DECL_AND_CODE	Indicates support for mixed declaration and code.	extc1x | stdc99 | extc99 | extc89 | extended
__C99_MIXED_STRING_CONCAT	Indicates support for concatenation of wide string and non-wide string literals.	extc1x | stdc99 | extc99 | extc89 | extended
__C99_NON_LVALUE_ARRAY_SUB	Indicates support for non-lvalue subscripts for arrays.	extc1x | stdc99 | extc99 | extc89 | extended
__C99_NON_CONST_AGGR_INITIALIZER	Indicates support for non-constant aggregate initializers.	extc1x | stdc99 | extc99 | extc89 | extended
__C99_PRAGMA_OPERATOR	Indicates support for the `_Pragma` operator.	extc1x | stdc99 | extc99 | extc89 | extended
__C99_REQUIRE_FUNC_DECL	Indicates that implicit function declaration is not supported.	stdc99
__C99_RESTRICT	Indicates support for the C99 `restrict` qualifier.	

Table 39. Predefined macros for language features (continued)

Predefined macro name	Description	Predefined when the following language level is in effect
__C99_STATIC_ARRAY_SIZE	Indicates support for the `static` keyword in array parameters to functions.	extc1x | stdc99 | extc99 | extc89 | extended
__C99_STD_PRAGMAS	Indicates support for standard pragmas.	extc1x | stdc99 | extc99 | extc89 | extended
__C99_TGMATH	Indicates support for type-generic macros in tgmath.h	extc1x | stdc99 | extc99 | extc89 | extended
__C99_UCN	Indicates support for universal character names.	extc1x | stdc99 | extc99 | ucs
__C99_VAR_LEN_ARRAY	Indicates support for variable length arrays.	extc1x | stdc99 | extc99 | extc89 | extended
__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.)
__DIGRAPHS__	Indicates support for digraphs.	extc1x | stdc99 | extc99 | extc89 | extended (also **-qdigraph**)
__EXTENDED__	Indicates that language extensions are supported.	extended
__IBM__ALIGN	Indicates support for the `__align` type qualifier.	Always defined except when **-qnokeyword=__alignof** is specified
__IBM_ALIGNOF__, __IBM__ALIGNOF__	Indicates support for the `__alignof__` operator.	extc1x | extc99 | extc89 | extended
__IBM_ATTRIBUTES	Indicates support for type, variable, and function attributes.	extc1x | extc99 | extc89 | extended
__IBM_COMPUTED_GOTO	Indicates support for computed `goto` statements.	extc1x | extc99 | extc89 | extended
__IBM_EXTENSION_KEYWORD	Indicates support for the `__extension__` keyword.	extc1x | extc99 | extc89 | extended
__IBM_GCC__INLINE__	Indicates support for the GCC `__inline__` specifier.	extc1x | extc99 | extc89 | extended
__IBM_DOLLAR_IN_ID	Indicates support for dollar signs in identifiers.	extc1x | extc99 | extc89 | extended
__IBM_GENERALIZED_LVALUE	Indicates support for generalized lvalues.	extc1x | extc99 | extc89 | extended
__IBM_INCLUDE_NEXT	Indicates support for the `#include_next` preprocessing directive.	Always defined
__IBM_LABEL_VALUE	Indicates support for labels as values.	extc1x | extc99 | extc89 | extended
__IBM_LOCAL_LABEL	Indicates support for local labels.	extc1x | extc99 | extc89 | extended

Table 39. Predefined macros for language features (continued)

Predefined macro name	Description	Predefined when the following language level is in effect
__IBM_MACRO_WITH_VA_ARGS	Indicates support for variadic macro extensions.	extc1x | extc99 | extc89 | extended
__IBM_NESTED_FUNCTION	Indicates support for nested functions.	extc1x | extc99 | extc89 | extended
__IBM_PP_PREDICATE	Indicates support for `#assert`, `#unassert`, `#cpu`, `#machine`, and `#system` preprocessing directives.	extc1x | extc99 | extc89 | extended
__IBM_PP_WARNING	Indicates support for the `#warning` preprocessing directive.	extc1x | extc99 | extc89 | extended
__IBM_REGISTER_VARS	Indicates support for variables in specified registers.	Always defined.
__IBM__TYPEOF__	Indicates support for the `__typeof__` or `typeof` keyword.	Always defined
__IBMC_COMPLEX_INIT	Indicates support for the macro based initialization of complex types: float _Complex, double _Complex, and long double _Complex.	extc1x
__IBMC_GENERIC	Indicates support for the generic selection feature.	extc89 | extc99 | extended | extc1x
__IBMC_NORETURN	Indicates support for the `_Noreturn` function specifier.	extc89 | extc99 | extended | extc1x extended | extended0x | c1xnoreturn
__IBMC_STATIC_ASSERT	Indicates support for the static assertions feature.	extc89 | extc99 | extended | extc1x
_LONG_LONG	Indicates support for `long long` data types.	extc1x | stdc99 | extc99 | | stdc89 | extc89 | extended (also -qlonglong)
__SAA__	Indicates that only language constructs that support the most recent level of SAA C standards are allowed.	saa
__SAA_L2__	Indicates that only language constructs that conform to SAA Level 2 C standards are allowed.	saal2
__STDC__	Indicates that the compiler conforms to the ANSI/ISO C standard.	Predefined to 1 if ANSI/ISO C standard conformance is in effect.

Table 39. Predefined macros for language features (continued)

Predefined macro name	Description	Predefined when the following language level is in effect
__STDC_HOSTED__	Indicates that the implementation is a hosted implementation of the ANSI/ISO C standard. (That is, the hosted environment has all the facilities of the standard C available).	extc1x | stdc99 | extc99
__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.)

Examples of predefined macros

This example illustrates use of the __FUNCTION__ and the __C99__FUNC__ macros to test for the availability of the C99 `__func__` identifier to return the current function name:

```
#include <stdio.h>

#if defined(__C99__FUNC__)
#define PRINT_FUNC_NAME() printf (" In function %s \n", __func__);
#elif defined(__FUNCTION__)
#define PRINT_FUNC_NAME() printf (" In function %s \n", __FUNCTION__);
#else
#define PRINT_FUNC_NAME() printf (" Function name unavailable\n");
#endif

void foo(void);

int main(int argc, char **argv)
{
   int k = 1;
   PRINT_FUNC_NAME();
   foo();
   return 0;
}

void foo (void)
{
   PRINT_FUNC_NAME();
   return;
}
```

The output of this example is:

```
In function main
In function foo
```

Chapter 7. Compiler built-in functions

A built-in function is a coding extension to 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 language. 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, V12.1. Furthermore, the technique can be time-consuming to implement.

As an alternative to managing hardware registers through assembly language, XL C built-in functions provide access to the optimized Power instruction set and allow the compiler to optimize the instruction scheduling.

The following sections describe the available built-in functions for the AIX platform.

Fixed-point built-in functions

Fixed-point built-in functions are grouped into the following categories:

- “Absolute value functions”
- “Assert functions” on page 414
- “Count zero functions” on page 415
- “Load functions” on page 417
- “Multiply functions” on page 417
- “Population count functions” on page 418
- “Rotate functions” on page 419
- “Store functions” on page 420
- “Trap functions” on page 421

Absolute value functions

__labs, __llabs

Purpose

Absolute Value Long, Absolute Value Long Long

Returns the absolute value of the argument.

Prototype

```
signed long __labs (signed long);

signed long long __llabs (signed long long);
```

Assert functions

__assert1, __assert2

Purpose

Generates trap instructions.

Prototype

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

void __assert2 (int);
```

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.

Valid only when **-qarch** is set to target POWER7 processors or higher in 64-bit mode.

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);
```

Usage

Valid only when **-qarch** is set to target POWER6 processors or higher.

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

These division functions are valid only when **-qarch** is set to target POWER7 processors or higher.

__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

Valid only when **-qarch** is set to target POWER7 processors or higher in 64-bit mode.

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

Valid only when **-qarch** is set to target POWER7 processors or higher in 64-bit mode.

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

Valid only when **-qarch** is set to target POWER7 processors or higher.

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

Valid only when **-qarch** is set to target POWER7 processors or higher.

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);

Usage

Valid only when **-qarch** is set to target POWER7 or higher processors in 64-bit mode.

Multiply functions

__imul_dbl

Purpose

Computes the product of two `long` integers and stores the result in a pointer.

Prototype

void __imul_dbl (long, long, long*);

__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);

Usage

Valid only in 64-bit mode.

__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);

Usage

Valid only when **-qarch** is set to target POWER7 processors or higher in 64-bit mode.

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.

Usage

`__tdw` is valid only in 64-bit mode.

__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);

Usage

`__trapd` is valid only in 64-bit mode.

Binary floating-point built-in functions

Binary floating-point built-in functions are grouped into the following categories:

- "Absolute value functions" on page 413
- "Add functions"
- "Conversion functions" on page 423
- "FPSCR functions" on page 425
- "Multiply functions" on page 428
- "Multiply-add/subtract functions" on page 428
- "Reciprocal estimate functions" on page 429
- "Rounding functions" on page 430
- "Select functions" on page 431
- "Square root functions" on page 431
- "Software division functions" on page 432

For decimal floating-point built-in functions, see Decimal floating-point built-in functions.

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);
```

Add functions

__fadd, __fadds

Purpose

Floating Add, Floating Add Single

Adds two arguments and returns the result.

Prototype

double __fadd (double, double);

float __fadds (float, 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;
}
```

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

__dfp_set_rounding_mode

Purpose

Set Rounding Mode

Sets the current decimal rounding mode.

Prototype

void __dfp_set_rounding_mode (unsigned long *rounding_mode*);

Parameters

`rounding_mode`
: One of the compile-time constant values (0 to 7) or macros listed in Table 41 on page 448.

Usage

If you change the rounding mode within a function, you must restore the rounding mode before the function returns.

__dfp_get_rounding_mode

Purpose

Get Rounding Mode

Gets the current decimal rounding mode.

Prototype

```
unsigned long __dfp_get_rounding_mode (void);
```

Return value

The current rounding mode as one of the values (0 to 7) listed in Table 41 on page 448.

Multiply functions

__fmul, __fmuls

Purpose

Floating Multiply, Floating Multiply Single

Multiplies two arguments and returns the result.

Prototype

```
double __fmul (double, double);

float __fmuls (float, float);
```

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 431.

__fre, __fres

Purpose

Floating Reciprocal Estimate, Floating Reciprocal Estimate Single

Prototype

double __fre (double);

float __fres (float);

Usage

`__fre` is valid only when **-qarch** is set to target POWER5 or later processors.

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);
```

Usage

Valid only when **-qarch** is set to target POWER5+ or later processors.

__frin, __frins

Purpose

Floating Round to Integer Nearest

Rounds the floating-point argument to an integer using round-to-nearest mode, and returns the value as a floating-point value.

Prototype

```
double __frin (double);

float __frins (float);
```

Usage

Valid only when **-qarch** is set to target POWER5+ or later processors.

__frip, __frips

Purpose

Floating Round to Integer Plus

Rounds the floating-point argument to an integer using round-to-plus-infinity mode, and returns the value as a floating-point value.

Prototype

double __frip (double);

float __frips (float);

Usage

Valid only when **-qarch** is set to target POWER5+ or later processors.

__friz, __frizs

Purpose

Floating Round to Integer Zero

Rounds the floating-point argument to an integer using round-to-zero mode, and returns the value as a floating-point value.

Prototype

double __friz (double);

float __frizs (float);

Usage

Valid only when **-qarch** is set to target POWER5+ or later processors.

Select functions

__fsel, __fsels

Purpose

Floating Select, Floating Select Single

Returns the second argument if the first argument is greater than or equal to zero; returns the third argument otherwise.

Prototype

double __fsel (double, double, double);

float __fsels (float, float, float);

Square root functions

__frsqrte, __frsqrtes

Purpose

Floating Reciprocal Square Root Estimate, Floating Reciprocal Square Root Estimate Single

Prototype

double __frsqrte (double);

float __frsqrtes (float);

Usage

`__frsqrtes` is valid only when **-qarch** is set to target POWER5+ or later processors.

__fsqrt, __fsqrts

Purpose

Floating Square Root, Floating Square Root Single

Prototype

double __fsqrt (double);

float __fsqrts (float);

Software division functions

__swdiv, __swdivs

Purpose

Software Divide, Software Divide Single

Divides the first argument by the second argument and returns the result.

Prototype

double __swdiv (double, double);

float __swdivs (float, float);

__swdiv_nochk, __swdivs_nochk

Purpose

Software Divide No Check, Software Divide No Check Single

Divides the first argument by the second argument, without performing range checking, and returns the result.

Prototype

double __swdiv_nochk (double *a*, double *b*);

float __swdivs_nochk (float *a*, float *b*);

Parameters

a Must not equal infinity. When **-qstrict** is in effect, *a* must have an absolute value greater than 2^{-970} and less than infinity.

b Must not equal infinity, zero, or denormalized values. When **-qstrict** is in effect, *b* must have an absolute value greater than 2^{-1022} and less than 2^{1021}.

Return value

The result must not be equal to positive or negative infinity. When **-qstrict** in effect, the result must have an absolute value greater than 2^{-1021} and less than 2^{1023}.

Usage

This function can provide better performance than the normal divide operator or the `__swdiv` built-in function in situations where division is performed repeatedly in a loop and when arguments are within the permitted ranges.

Store functions

__stfiw

Purpose

Store Floating Point as Integer Word

Stores the contents of the loworder 32 bits of *value*, without conversion, into the word in storage addressed by *addr*.

Prototype

void __stfiw (const int* *addr*, double *value*);

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

The following functions are valid only when **-qarch** is set to target POWER8 processors:

- “__bcdadd” on page 434
- “__bcdsub” on page 434

__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

The following functions are valid only when **-qarch** is set to target POWER8 processors:

- "__bcdadd_ofl"
- "__bcdsub_ofl" on page 435
- "__bcd_invalid" on page 435

__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

The following functions are valid only when **-qarch** is set to target POWER8 processors:

- "__bcdcmpeq"
- "__bcdcmpge"
- "__bcdcmpgt"
- "__bcdcmple" on page 436
- "__bcdcmplt" on page 436

__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

The following functions are valid only when **-qarch** is set to target POWER7 or POWER8 processors:

- "__vec_ldrmb"
- "__vec_strmb"

__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.

Decimal floating-point built-in functions

Decimal floating-point (DFP) built-in functions are grouped into the following categories:

- "Absolute value functions"
- "Coefficient functions" on page 438
- "Comparison functions" on page 439
- "Conversion functions" on page 440
- "Exponent functions" on page 445
- "NaN functions" on page 446
- "Register transfer functions" on page 447
- "Rounding functions" on page 448
- "Test functions" on page 450

For binary floating-point built-in functions, see Binary floating-point built-in functions

When **-qarch** is set to **pwr6**, **pwr6e**, or later POWER processors, **-qfloat=nodfpemulate** becomes the default. This means that DFP hardware instructions are generated. Lower-performance software emulation code is generated only when:

- **-qarch** is set to **pwr5**.
- **-qarch** is set to **pwr6**, **pwr6e**, or later processors, and **-qfloat=dfpemulate** is enabled

Absolute value functions

Absolute value functions determine the sign of the returned value.

__d64_abs, __d128_abs

Purpose

Absolute Value

Returns the absolute value of the parameter.

Prototype

```
_Decimal64 __d64_abs (_Decimal64);

_Decimal128 __d128_abs (_Decimal128);
```

__d64_nabs, __d128_nabs

Purpose

Negative Absolute Value

Returns the negative absolute value of the parameter.

Prototype

_Decimal64 __d64_nabs (_Decimal64);

_Decimal128 __d128_nabs (_Decimal128);

__d64_copysign, __d128_copysign

Purpose

Copysign

Returns the absolute value of the first parameter, with the sign of the second parameter.

Prototype

_Decimal64 __d64_copysign (_Decimal64, _Decimal64);

_Decimal128 __d128_copysign (_Decimal128, _Decimal128);

Coefficient functions

Coefficient functions manipulate the fraction without affecting the exponent and sign, to support decimal-floating point conversion library functions.

__d64_shift_left, __d128_shift_left

Purpose

Shift Coefficient Left.

Shifts the coefficient of the parameter left.

Prototype

_Decimal64 __d64_shift_left (_Decimal64, unsigned long *digits*);

_Decimal128 __d128_shift_left (_Decimal128, unsigned long *digits*);

Parameters

digits
: The number of digits to be shifted left. The shift count must be in the range 0 to 63; otherwise the result is undefined.

Return value

The sign and exponent are unchanged. The digits are shifted left.

__d64_shift_right, __d128_shift_right

Purpose

Shift Coefficient Right.

Shifts the coefficient of the parameter right.

Prototype

_Decimal64 __d64_shift_right (_Decimal64, unsigned long *digits*);

_Decimal128 __d128_shift_right (_Decimal128, unsigned long *digits*);

Parameters

digits
The number of digits to be shifted right. The shift count must be in the range 0 to 63; otherwise the result is undefined.

Return value

The sign and exponent are unchanged. The digits are shifted right.

Comparison functions

Comparison functions support extended exception handling and exponent comparisons.

__d64_compare_exponents, __d128_compare_exponents

Purpose

Compare Exponents

Compares the exponents of two decimal floating-point values.

Prototype

long __d64_compare_exponents (_Decimal64, _Decimal64);

long __d128_compare_exponents (_Decimal128, _Decimal128);

Return value

Returns the following values:

- Less than 0 if the exponent of the first parameter is less than the exponent of the second parameter.
- 0 if both parameters have the same exponent value or if both are quiet or signaling NaNs (quiet and signaling are considered equal) or both are infinities.
- Greater than 0 if the exponent of the first argument is greater than the exponent of the second argument.
- -2 if one of the two parameters is a quiet or signaling NaN or one of the two parameters is an infinity.

__d64_compare_signaling, __d128_compare_signaling

Purpose

Compare Signaling Exception on NaN

Compares two decimal floating-point values and raises an Invalid Operation exception if either is a quiet or signaling NaN.

Prototype

long __d64_compare_signaling (_Decimal64, _Decimal64);

long __d128_compare_signaling (_Decimal128, _Decimal128);

Return value

Returns the following values:

- Less than 0 if the value of the first parameter is less than the value of the second parameter.
- 0 if both parameters have the same value.
- Greater than 0 if the value of the first parameter is greater than the value of the second parameter.

If either value is a quiet or signalling NaN, an exception is raised. If no exception handler has been enabled to trap the exception, the function returns -2.

Usage

If either value is a NaN, normal comparisons using the relational operators (==, !=, <, <=, > and >=) always return false, which raises an exception for a signaling NaN but not for a quiet NaN. If you want an exception to be raised when either value is a quiet or signaling NaN, you should use the Compare Signaling Exception on NaN built-in functions instead of a relational operator.

Conversion functions

Conversion functions execute decimal floating-point conversions. Some override the current rounding mode.

__cbcdtd

Purpose

Convert Binary Coded Decimal to Declets.

The low-order 24 bits of each word of the source contain six, 4-bit BCD fields that are converted to two declets; each set of the two declets is placed into the low-order 20 bits of the corresponding word in the result. The high-order 12 bits in each word of the result are set to 0. If a 4-bit BCD field has a value greater than 9, the results are undefined.

Prototype

long long __cbcdtd (long long);

Usage

Valid only when **-qarch** is set to target POWER7 processors or higher.

__cdtbcd

Purpose

Convert Declets to Binary Coded Decimal.

The low-order 20 bits of each word of the source contain two declets that are converted to six, 4-bit BCD fields; each set of six, 4-bit BCD fields is placed into the low-order 24 bits of the corresponding word in the result. The high-order 8 bits in each word of the result are set to 0.

Prototype

long long __cdtbcd (long long);

Usage

Valid only when **-qarch** is set to target POWER7 processors or higher.

__d64_to_long_long, __d128_to_long_long

Purpose

Convert to Integer

Converts a decimal floating-point value to a 64-bit signed binary integer, using the current rounding mode.

Prototype

long long __d64_to_long_long (_Decimal64);

long long __d128_to_long_long (_Decimal128):

Return value

The input value converted to a long long, using the current rounding mode (not always rounded towards zero as a cast or implicit conversion would be).

__d64_to_long_long_rounding, __d128_to_long_long_rounding

Purpose

Convert to Integer

Converts a decimal floating-point value to a 64-bit signed binary integer, using a specified rounding mode.

Prototype

long long __d64_to_long_long_rounding (_Decimal64, long *rounding_mode*);

long long __d128_to_long_long_rounding (_Decimal128, long *rounding_mode*);

Parameters

mode
: One of the compile time constant values or macros defined in Table 41 on page 448.

Return value

The input value converted to a long long, using the specified rounding mode (not always rounded towards zero as a cast or implicit conversion would be).

Usage

These functions temporarily override the rounding mode in effect for the current operation.

__d64_to_signed_BCD

Purpose

Convert to Signed Binary-Coded Decimal

Converts the lower digits of a 64-bit decimal floating-point value to a Signed Packed Format (packed decimal).

Prototype

> unsigned long long __d64_to_signed_BCD (_Decimal64, _Bool *value*);

Return value

Produces 15 decimal digits followed by a decimal sign in a 64-bit result. The leftmost digit is ignored.

Positive values are given the sign 0xF if *value* is true and 0xC if *value* is false.

Negative values are given the sign 0xD.

Usage

You can use the `__d64_shift_right` function to access the leftmost digit.

__d128_to_signed_BCD

Purpose

Convert to Signed Binary Coded Decimal.

Converts the lower digits of a 128-bit decimal floating-point value to a Signed Packed Format (packed decimal).

Prototype

> void __d128_to_signed_BCD (_Decimal128, _Bool *value*, unsigned long long **upper*, unsigned long long **lower*);

Parameters

upper
: The address of the variable that will hold the upper digits of the result.

lower
: The address of the variable that will hold the lower digits of the result.

Return value

Produces 31 decimal digits followed by a decimal sign in a 128-bit result. Digits to the left are ignored. The higher 16 digits are stored in the parameter *upper*. The lower 15 digits plus the sign are stored in the parameter *lower*.

Positive values are given the sign 0xF if *value* is true and 0xC if *value* is false.

Negative values are given the sign 0xD.

Usage

You can use the `__d128_shift_right` function to access the digits to the left.

__d64_to_unsigned_BCD

Purpose

Convert to Unsigned Binary Coded Decimal.

Converts the lower digits of a 64-bit decimal floating-point value to an Unsigned Packed Format.

Prototype

unsigned long long __d64_to_unsigned_BCD (_Decimal64);

Return value

Returns 16 decimal digits with no sign in a 64-bit result.

Usage

You can use the `__d64_shift_right` function to access the digits to the left.

__d128_to_unsigned_BCD

Purpose

Convert to Unsigned Binary Coded Decimal.

Converts the lower digits of a 128-bit decimal floating-point value to an Unsigned Packed Format.

Prototype

void __d128_to_unsigned_BCD (_Decimal128, unsigned long long **upper*, unsigned long long **lower*);

Parameters

`upper`
The address of the variable that will hold the upper digits of the result.

`lower`
The address of the variable that will hold the lower digits of the result.

Return value

Produces 32 decimal digits with no sign in a 128-bit result. Digits to the left are ignored. The higher 16 digits are stored in the parameter *upper*. The lower 16 digits are stored in the parameter *lower*.

Usage

You can use the `__d128_shift_right` function to access the digits to the left.

__signed_BCD_to_d64

Purpose

Convert from Signed Binary Coded Decimal.

Converts a 64-bit Signed Packed Format (packed decimal - 15 decimal digits followed by a decimal sign) to a 64-bit decimal floating-point value.

Prototype

_Decimal64 __signed_BCD_to_d64 (unsigned long long);

Parameters

The signs 0xA, 0xC, 0xE, and 0xF are treated as positive. The signs 0xB and 0xD are treated as negative.

__signed_BCD_to_d128

Purpose

Convert from Signed Binary Coded Decimal.

Converts a 128-bit Signed Packed Format (packed decimal - 31 decimal digits followed by a decimal sign) to a 128-bit decimal floating-point value.

Prototype

_Decimal128 __signed_BCD_to_d128 (unsigned long long *upper*, unsigned long long *lower*);

Parameters

upper
The upper 16 digits of the input value.

lower
The lower 15 digits plus the sign of the input value.

Parameters

The signs 0xA, 0xC, 0xE, and 0xF are treated as positive. The signs 0xB and 0xD are treated as negative.

__unsigned_BCD_to_d64

Purpose

Convert from Unsigned Binary Coded Decimal.

Converts a 64-bit Unsigned Packed Format (16 decimal digits with no sign) to a 64-bit decimal floating-point value.

Prototype

_Decimal64 __unsigned_BCD_to_d64 (unsigned long long);

__unsigned_BCD_to_d128

Purpose

Convert from Unsigned Binary Coded Decimal.

Converts a 128-bit Unsigned Packed Format (32 decimal digits with no sign) to a 128-bit decimal floating-point value.

Prototype

_Decimal128 __unsigned_BCD_to_d128 (unsigned long long *upper*, unsigned long long *lower*);

Parameters

upper
The upper 16 digits of the input value.

lower
The lower 16 digits of the input value.

Exponent functions

Exponent functions extract the exponent from a value or insert an exponent into a value, primarily to support decimal-floating point conversion library functions. They use special values to identify or specify the exponent type.

Table 40. Biased exponents macros and values

Macro	Integer value
DFP_BIASED_EXPONENT_FINITE	0
DFP_BIASED_EXPONENT_INFINITY	-1
DFP_BIASED_EXPONENT_QNAN	-2
DFP_BIASED_EXPONENT_SNAN	-3

__d64_biased_exponent, __d128_biased_exponent

Purpose

Extract Biased Exponent

Returns the exponent of a decimal floating-point value as an integer.

Prototype

long __d64_biased_exponent (_Decimal64);

long __d128_biased_exponent (_Decimal128);

Return value

Returns special values for infinity, quiet NaN, and signalling NaN, as listed in Table 40.

For finite values, the result is DFP_BIASED_EXPONENT_FINITE plus the exponent bias (398 for _Decimal64, 6176 for _Decimal128) plus the actual exponent.

__d64_insert_biased_exponent, __d128_insert_biased_exponent

Purpose

Insert Biased Exponent

Replaces the exponent of a decimal floating-point value.

Prototype

_Decimal64 __d64_insert_biased_exponent (_Decimal64, long *exponent*);

_Decimal128 __d128_insert_biased_exponent (_Decimal128, long *exponent*);

Parameters

exponent
The exponent value to be applied to the first parameter. For infinity, quiet NaN and signalling NaN, use one of the compile-time constant values or macros listed in Table 40 on page 445.

For finite values, the result is DFP_BIASED_EXPONENT_FINITE plus the exponent bias (398 for _Decimal64, 6176 for _Decimal128) plus the corresponding exponent.

NaN functions

NaN functions create quiet or signaling NaNs.

__d32_sNaN, __d64_sNaN, __d128_sNaN

Purpose

Make Signalling NaN

Creates a signalling NaN of the specified precision, with a positive sign and zero payload.

Prototype

_Decimal32 __d32_sNan (void);

_Decimal64 __d64_sNaN (void);

_Decimal128 __d128_sNaN (void);

__d32_qNaN, __d64_qNaN, __d128qNaN

Purpose

Make Quiet NaN

Creates a quiet NaN of the specified precision, with a positive sign and zero payload.

Prototype

_Decimal32 __d32_qNaN (void);

_Decimal64 __d64_qNaN (void);

_Decimal128 __d128_qNaN (void);

Register transfer functions

Register transfer functions transfer data between general purpose registers and floating-point registers. No conversion occurs. Register transfer functions handle integer data in floating-point registers or floating-point data in general purpose registers. These functions use instructions that are available with **-qarch=pwr6** or **-qarch=pwr6e** only, on a POWER6 running in POWER6e (raw) mode.

__gpr_to_d64

Purpose

Transfer from General Purpose Register to Floating-Point Register

Transfers a value from a general purpose register (64-bit mode) or a general purpose register pair (32-bit mode).

Prototype

_Decimal64 __gpr_to_d64 (long long);

__gprs_to_d128

Purpose

Transfer from General Purpose Register to Floating-Point Register.

Transfers a value from a pair of general purpose registers (64-bit mode) or four general purpose registers (32-bit mode).

Prototype

_Decimal128 __gprs_to_d128 (unsigned long long**upper*, unsigned long long**lower*);

Parameters

upper
The address of the variable that will hold the upper 64 bits of the result.

lower
The address of the variable that will hold the lower 64 bits of the result.

Return value

The higher 64 bits are stored in the parameter *upper*. The lower 64 bits are stored in the parameter *lower*.

__d64_to_gpr

Purpose

Transfer from Floating-Point Register to General Purpose Register.

Transfers a value from a floating-point register to a general purpose register (64-bit mode) or a general purpose register pair (32-bit mode).

Prototype

long long __d64_to_gpr (_Decimal64);

__d128_to_gprs

Purpose

Transfer from Floating-Point Register to General Purpose Register.

Transfers a value from a pair of floating-point registers to a pair of general purpose registers (64-bit mode) or four general purpose registers (32-bit mode).

Prototype

void __d128_to_gprs (_Decimal128, unsigned long long*_upper_, unsigned long long*_lower_);

Parameters

upper
The address of the variable that contains the upper 64 bits of the input value.

lower
The address of the variable that contains the lower 64 bits of the input value.

Rounding functions

Rounding functions perform operations such as rounding and truncation of floating-point values.

Table 41. Rounding mode macros and values

Macro	**Integer value**
DFP_ROUND_TO_NEAREST_WITH_TIES_TO_EVEN	0
DFP_ROUND_TOWARD_ZERO	1
DFP_ROUND_TOWARD_POSITIVE_INFINITY	2
DFP_ROUND_TOWARD_NEGATIVE_INFINITY	3
DFP_ROUND_TO_NEAREST_WITH_TIES_AWAY_FROM_ZERO	4
DFP_ROUND_TO_NEAREST_WITH_TIES_TOWARD_ZERO	5
DFP_ROUND_AWAY_FROM_ZERO	6
DFP_ROUND_TO_PREPARE_FOR_SHORTER_PRECISION	7
DFP_ROUND_USING_CURRENT_MODE[1]	8

Note:

1. This value is valid only for functions that override the current rounding mode; it is not valid for `__dfp_set_rounding_mode` and can not be returned by `__dfp_get_rounding_mode`.

__d64_integral, __d128_integral

Purpose

Round to Integral

Rounds a decimal floating-point value to an integer, allowing an Inexact exception to be raised.

Prototype

_Decimal64 __d64_integral (_Decimal64);

_Decimal128 __d128_integral (_Decimal128);

Return value

The integer is returned in decimal floating-point format, rounded using the current rounding mode. Digits after the decimal point are discarded.

__d64_integral_no_inexact, __d128_integral_no_inexact

Purpose

Round to Integral

Rounds a decimal floating-point value to an integer, suppressing any Inexact exception from being raised.

Prototype

_Decimal64 __d64_integral_no_inexact (_Decimal64);

_Decimal128 __d128_integral_no_inexact (_Decimal128);

Return value

The integer is returned in decimal floating-point format, rounded using the current rounding mode. Digits after the decimal point are discarded.

__d64_quantize, __d128_quantize

Purpose

Quantize

Returns the arithmetic value of the first parameter, with the exponent adjusted to match the second parameter, using a specified rounding mode.

Prototype

_Decimal64 __d64_quantize (_Decimal64, _Decimal64, long *rounding_mode*);

_Decimal128 __d128_quantize (_Decimal128, _Decimal128, long *rounding_mode*);

Parameters

rounding_mode
: One of the compile-time constant values or macros defined in Table 41 on page 448.

Usage

These functions temporarily override the rounding mode in effect for the current operation.

__d64_reround, __d128_reround

Purpose

Reround

Complete rounding of a partially rounded value, avoiding double rounding which causes errors.

Prototype

> _Decimal64 __d64_reround (_Decimal64, unsigned long *number_of_digits*, unsigned long *rounding_mode*);
>
> _Decimal128 __d128_reround (_Decimal128, unsigned long *number_of_digits*, unsigned long *rounding_mode*);

Parameters

number_of_digits
The number of digits to round to, from 1 to 15 for __d64_reround and from 1 to 33 for __d128_reround.

rounding_mode
One of the compile-time constant values or macros defined in Table 41 on page 448.

Usage

These functions temporarily override the rounding mode in effect for the current operation. The value to be rerounded should have been previously rounded using mode DFP_ROUND_TO_PREPARE_FOR_SHORTER_PRECISION or 7 to ensure correct rounding.

Test functions

Test functions allow extended exception handling of invalid results or categorization of input values, primarily to support math library functions.

Those functions that begin with __d64_is or __d128_is will not raise an exception, even for signaling NaNs.

Table 42. Test data class mask macros and values

Macro	**Integer value**
DFP_PPC_DATA_CLASS_ZERO	0x20
DFP_PPC_DATA_CLASS_SUBNORMAL	0x10
DFP_PPC_DATA_CLASS_NORMAL	0x08
DFP_PPC_DATA_CLASS_INFINITY	0x04
DFP_PPC_DATA_CLASS_QUIET_NAN	0x02
DFP_PPC_DATA_CLASS_SIGNALING_NAN	0x01

Table 43. Test data group mask macros and values

Macro	Integer value
DFP_PPC_DATA_GROUP_SAFE_ZERO	0x20
DFP_PPC_DATA_GROUP_ZERO_WITH_EXTREME_EXPONENT	0x10
DFP_PPC_DATA_GROUP_NONZERO_WITH_EXTREME_EXPONENT	0x08
DFP_PPC_DATA_GROUP_SAFE_NONZERO	0x04
DFP_PPC_DATA_GROUP_NONZERO_LEFTMOST_DIGIT_NONEXTREME_EXPONENT	0x02
DFP_PPC_DATA_GROUP_SPECIAL	0x01

Table 44. Test data class and group result macros and values

Macro	Integer value
DFP_PPC_DATA_POSITIVE_NO_MATCH	0x00
DFP_PPC_DATA_POSITIVE_MATCH	0x02
DFP_PPC_DATA_NEGATIVE_NO_MATCH	0x08
DFP_PPC_DATA_NEGATIVE_MATCH	0x0A

Table 45. Test data class and group result mask macros and values

Macro	Integer value
DFP_PPC_DATA_NEGATIVE_MASK	0x08
DFP_PPC_DATA_MATCH_MASK	0x02

__d64_same_quantum, __d128_same_quantum

Purpose

Same Quantum

Returns true if two values have the same quantum

Prototype

```
_Bool __d64_same_quantum (_Decimal64, _Decimal64);

_Bool __d128_same_quantum (_Decimal28, _Decimal128);
```

__d64_issigned, __d128_issigned

Purpose

Is Signed

Returns true if the parameter is negative, negative zero, negative infinity, or negative NaN.

Prototype

```
_Bool __d64_issigned (_Decimal64);

_Bool __d128_issigned (_Decimal128);
```

__d64_isnormal, __d128_isnormal

Purpose

Is Normal

Returns true if the parameter is in the normal range (that is, not a subnormal, infinity or NaN) and not zero.

Prototype

```
_Bool _d64_isnormal (_Decimal64);

_Bool _d128_isnormal (_Decimal128);
```

__d64_isfinite, __d128_isfinite

Purpose

Is Finite

Returns true if the parameter is not positive or negative infinity and not a quiet or signaling NaN.

Prototype

```
_Bool __d64_isfinite (_Decimal64);

_Bool __d128_isfinite (_Decimal128);
```

__d64_iszero, __d128_iszero

Purpose

Is Zero

Returns true if the parameter is positive or negative zero.

Prototype

```
_Bool __d64_iszero (_Decimal64);

_Bool __d128_iszero (_Decimal128);
```

__d64_issubnormal, __d128_issubnormal

Purpose

Is Subnormal

Returns true if the parameter is a subnormal.

Prototype

```
_Bool _d64_issubnormal (_Decimal64);

_Bool _d128_issubnormal (_Decimal128);
```

__d64_isinf, __d128_isinf

Purpose

Is Infinity

Returns true if the parameter is positive or negative infinity.

Prototype

_Bool __d64_isinf (_Decimal64);

_Bool __d128_isinf (_Decimal128);

__d64_isnan, __d128_isnan

Purpose

Is NaN

Returns true if the parameter is a positive or negative quiet or signaling NaN.

Prototype

_Bool __d64_isnan (_Decimal64);

_Bool __d128_isnan (_Decimal128);

__d64_issignaling, __d128_issignaling

Purpose

Is Signaling NaN

Returns true if the parameter is a positive or negative signaling NaN.

Prototype

_Bool __d64_issignaling (_Decimal64);

_Bool __d128_issignaling (_Decimal128);

__d64_test_data_class, __d128_test_data_class

Purpose

Test Data Class

Reports if a value is a zero, subnormal, normal, infinity, quiet NaN or signaling NaN, and if the value is positive or negative.

Prototype

long __d64_test_data_class (_Decimal64, unsigned long *mask*);

long __d128_test_data_class (_Decimal128, unsigned long *mask*);

Parameters

mask
: One of the values or macros defined in Table 42 on page 450 or several ORed together. The parameter must be a compile time constant expression.

Return value

One of the values listed in Table 44 on page 451.

Usage

You can use an appropriate mask to check combinations of values at the same time. Use the masks listed in Table 42 on page 450 to check input values. Use the masks listed in Table 45 on page 451 to check result values.

__d64_test_data_group, __d128_test_data_group

Purpose

Test Data Group

Reports if a value is a safe zero, a zero with an extreme exponent, a subnormal, a safe nonzero, a normal with no leading zero, or an infinity or NaN and if the value is positive or negative. Safe means leading zero digits and a non-extreme exponent. A subnormal can appear as either an extreme nonzero or safe nonzero. The exact meaning of some masks depends on the particular CPU model.

Prototype

long _d64_test_data_group (_Decimal64, unsigned long *mask*);

long _d128_test_data_group (_Decimal128, unsigned long *mask*);

Parameters

mask
: One of the values or macros defined in Table 43 on page 451 or several ORed together. The parameter must be a compile time constant expression.

Return value

One of the values listed in Table 44 on page 451.

Usage

You can use an appropriate mask to check combinations of values at the same time. Use the masks listed in Table 43 on page 451 to check input values. Use the masks listed in Table 45 on page 451 to check result values.

__d64_test_significance, __d128_test_significance

Purpose

Test Significance

Checks whether a decimal floating-point value has a specified number of digits of significance.

Prototype

long __d64_test_significance (_Decimal64, unsigned long *digits*);

long __d128_test_significance (_Decimal128, unsigned long *digits*);

Parameters

digits
The number of digits of significance to be tested for. *digits* must be in the range 0 to 63; otherwise the result is undefined. If it is 0, all values including zero will be considered to have more significant digits, if it is not 0, a zero value will be considered to have fewer significant digits.

Return value

Returns the following values:

- Less than 0 if the number of digits of significance of the first parameter is less than the second parameter.
- 0 if the number of digits of significance is the same as the second parameter.
- Greater than 0 if the number of digits of significance of the first parameter is greater than that of the second parameter or digits is 0.
- -2 if either parameter is a quiet or signaling NaN or positive or negative infinity.

For these functions, the number of significant digits of the value 0 is considered to be zero.

Miscellaneous functions

This section lists the miscellaneous decimal floating-point built-in functions.

__addg6s

Purpose

Add and Generate Sixes

Adds *source1* to *source2* and produces 16 carry bits, one for each carry out of decimal position *n* (bit position 4xn).

The result is a doubleword composed of the 16 carry bits. The doubleword consists of a decimal six (0b0110) in every decimal digit position for which the corresponding carry bit is 0, and a zero (0b0000) in every position for which the corresponding carry bit is 1.

Prototype

long long __addg6s (long long source1, long long source2);

Usage

Valid only when **-qarch** is set to target POWER7 processors or higher in 64-bit mode.

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 457
- "Compare and swap functions" on page 458
- "Fetch functions" on page 459
- "Load functions" on page 461
- "Store functions" on page 462
- "Synchronization functions" on page 462

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.

Usage

`__check_lockd_mp` is valid only in 64-bit mode.

__check_lock_up, __check_lockd_up

Purpose

Check Lock on Uniprocessor Systems, Check Lock Doubleword on Uniprocessor Systems

Conditionally updates a single word or doubleword variable atomically.

Prototype

unsigned int __check_lock_up (const int* *addr*, int *old_value*, int *new_value*);

unsigned int __check_lockd_up (const long* *addr*, long *old_value*, long *new_value*);

Parameters

addr
The address of the variable to be updated. Must be aligned on a 4-byte boundary for a single word and on an 8-byte boundary for a doubleword.

old_value
The old value to be checked against the current value in *addr*.

new_value
The new value to be conditionally assigned to the variable in *addr*,

Return value

Returns false (0) if the value in *addr* was equal to *old_value* and has been set to the new value. Returns true (1) if the value in *addr* was not equal to *old_value* and has been left unchanged.

Usage

`__check_lockd_up` is valid only in 64-bit mode.

Clear lock functions

__clear_lock_mp, __clear_lockd_mp

Purpose

Clear Lock on Multiprocessor Systems, Clear Lock Doubleword on Multiprocessor Systems

Atomic store of the *value* into the variable at the address *addr*.

Prototype

void __clear_lock_mp (const int* *addr*, int *value*);

void __clear_lockd_mp (const long* *addr*, long *value*);

Parameters

addr
The address of the variable to be updated. Must be aligned on a 4-byte boundary for a single word and on an 8-byte boundary for a doubleword.

value
The new value to be assigned to the variable in *addr*,

Usage

`__clear_lockd_mp` is only valid in 64-bit mode.

__clear_lock_up, __clear_lockd_up

Purpose

Clear Lock on Uniprocessor Systems, Clear Lock Doubleword on Uniprocessor Systems

Atomic store of the *value* into the variable at the address *addr*.

Prototype

void __clear_lock_up (const int* *addr*, int *value*);

void __clear_lockd_up (const long* *addr*, long *value*);

Parameters

addr
The address of the variable to be updated. Must be aligned on a 4-byte boundary for a single word and on an 8-byte boundary for a doubleword.

value
The new value to be assigned to the variable in *addr*.

Usage

`__clear_lockd_up` is only valid in 64-bit mode.

Compare and swap functions

__compare_and_swap, __compare_and_swaplp

Purpose

Conditionally updates a single word or doubleword variable atomically.

Prototype

int __compare_and_swap (volatile int* *addr*, int* *old_val_addr*, int *new_val*);

int __compare_and_swaplp (volatile long* *addr*, long* *old_val_addr*, long *new_val*);

Parameters

addr
The address of the variable to be copied. Must be aligned on a 4-byte boundary for a single word and on an 8-byte boundary for a doubleword.

old_val_addr
The memory location into which the value in *addr* is to be copied.

`new_val`
: The value to be conditionally assigned to the variable in *addr*,

Return value

Returns true (1) if the value in *addr* was equal to *old_value* and has been set to the new value. Returns false (0) if the value in *addr* was not equal to *old_value* and has been left unchanged. In either case, the contents of the memory location specified by *addr* are copied into the memory location specified by *old_val_addr*.

Usage

The `__compare_and_swap` function is useful when a single word value must be updated only if it has not been changed since it was last read. If you use `__compare_and_swap` as a locking primitive, insert a call to the `__isync` built-in function at the start of any critical sections.

`__compare_and_swaplp` is valid only in 64-bit mode.

Fetch functions

__fetch_and_and, __fetch_and_andlp

Purpose

Clears bits in the word or doubleword specified by*addr* by AND-ing that value with the value specified by *val*, in a single atomic operation, and returns the original value of *addr*.

Prototype

unsigned int __fetch_and_and (volatile unsigned int* *addr*, unsigned int *val*);

unsigned long __fetch_and_andlp (volatile unsigned long* *addr*, unsigned long *val*);

Parameters

`addr`
: The address of the variable to be ANDed. Must be aligned on a 4-byte boundary for a single word and on an 8-byte boundary for a doubleword.

`value`
: The value by which the value in *addr* is to be ANDed.

Usage

This operation is useful when a variable containing bit flags is shared between several threads or processes.

`__fetch_and_andlp` is valid only in 64-bit mode.

__fetch_and_or, __fetch_and_orlp

Purpose

Sets bits in the word or doubleword specified by *addr* by OR-ing that value with the value specified *val*, in a single atomic operation, and returns the original value of *addr*.

Prototype

unsigned int __fetch_and_or (volatile unsigned int* *addr*, unsigned int *val*);

unsigned long __fetch_and_orlp (volatile unsigned long* *addr*, unsigned long *val*);

Parameters

addr
: The address of the variable to be ORed. Must be aligned on a 4-byte boundary for a single word and on an 8-byte boundary for a doubleword.

value
: The value by which the value in *addr* is to be ORed.

Usage

This operation is useful when a variable containing bit flags is shared between several threads or processes.

`__fetch_and_orlp` is valid only in 64-bit mode.

__fetch_and_swap, __fetch_and_swaplp

Purpose

Sets the word or doubleword specified by *addr* to the value of *val* and returns the original value of *addr*, in a single atomic operation.

Prototype

unsigned int __fetch_and_swap (volatile unsigned int* *addr*, unsigned int *val*);

unsigned long __fetch_and_swaplp (volatile unsigned long* *addr*, unsigned long *val*);

Parameters

addr
: The address of the variable to be updated. Must be aligned on a 4-byte boundary for a single word and on an 8-byte boundary for a doubleword.

value
: The value which is to be assigned to *addr*.

Usage

This operation is useful when a variable is shared between several threads or processes, and one thread needs to update the value of the variable without losing the value that was originally stored in the location.

`__fetch_and_swaplp` is valid only in 64-bit mode.

Load functions

__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`, in 64-bit mode, 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 607 for a description of the `__fence` built-in function).

`__ldarx` and `__lqarx` are valid only in 64-bit mode. `__lqarx`, `__lharx`, and `__lbarx` are valid only when **-qarch** is set to target POWER8 processors.

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.

`__stdcx` is valid only in 64-bit mode. `__stqcx`, `__sthcx`, and `__stbcx` are valid only when **-qarch** is set to target POWER8 processors.

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 466

Data cache functions

__dcbf
Purpose

Data Cache Block Flush

Copies the contents of a modified block from the data cache to main memory and flushes the copy from the data cache.

Prototype

void __dcbf(const void* *addr*);

__dcbfl
Purpose

Data Cache Block Flush Line

Flushes the cache line at the specified address from the L1 data cache.

Prototype

void __dcbfl (const void* *addr*);

Usage

The target storage block is preserved in the L2 cache.

Valid when **-qarch** is set to target POWER6 processors or higher.

__dcbflp
Purpose

Data Cache Block Flush Line Primary

Flushes the cache line at address from the primary data cache of a single processor.

Prototype

void __dcbflp(const void* address);

Usage

Valid only when **-qarch** is set to target POWER7 processors or higher.

__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*);

Usage

Valid only when **-qarch** is set to target POWER8 processors.

__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) ;
```

Usage

Valid only when **-qarch** is set to target POWER8 processors.

Prefetch built-in functions

__dcbtstt

Purpose

Store Transient Touch provides a hint that describes a block that the program may perform a store access to. The block is likely to be transient, that is, the time interval during which the program accesses the unit is likely to be short.

Prototype

```
void __dcbtstt (void * address);
```

Usage

Valid only when **-qarch** is set to target POWER7 processors or higher.

__dcbtt

Purpose

Data Cache Block Touch Transient

Load Transient Touch provides a hint that describes a block that the program might perform a load access to. The block is likely to be transient, that is, the time interval during which the program accesses the unit is likely to be short.

Prototype

void __dcbtt (void * address);

Usage

Valid only when **-qarch** is set to target POWER7 processors or higher.

__partial_dcbt

Purpose

Partial Data Cache Block Touch

Loads half of the cache line that contains the specified address into the L3 data cache.

Prototype

void __partial_dcbt (void * address);

Usage

Valid only when **-qarch** is set to target POWER7 processors or higher.

__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*);

__protected_stream_count

Purpose

Sets the number of cache lines for a specific limited-length protected stream.

Prototype

void __protected_stream_count (unsigned int *unit_cnt*, unsigned int *stream_ID*);

Parameters

unit_cnt
The number of cache lines. Must be an integer with a value of 0 to 1023.

stream_ID
An integer with a value 0-7 on POWER5 processors, a value 0 to 15 on POWER6 processors, and a value 0 to 11 on POWER7 and POWER8 processors.

Usage

Valid only when **-qarch** is set to target POWER5 processors or higher.

__protected_stream_count_depth

Purpose

Sets the number of cache lines and the prefetch depth for a specific limited-length protected stream.

Prototype

> void _protected_stream_count_depth (unsigned int *unit_cnt*, unsigned int *prefetch_depth*, unsigned int *stream_ ID*);

Parameters

unit_cnt
The number of cache lines. Must be an integer with a value of 0 to 1023.

prefetch_depth
A relative, qualitative value which sets the steady-state *fetch-ahead* distance of the prefetches for a stream. The fetch-ahead distance is the number of lines being prefetched in advance of the line from which data is currently being loaded, or to which data is currently being stored. Valid values are as follows:

0 The default defined in the Data Stream Control Register.

1 None.

2 Shallowest.

3 Shallow.

4 Medium.

5 Deep.

6 Deeper.

7 Deepest.

stream_ID
An integer with a value 0 to 15 on POWER6 processors, and a value 0 to 11 on or POWER7 and POWER8 processors.

Usage

Valid only when **-qarch** is set to target POWER6 processors or higher.

__protected_stream_go

Purpose

Starts prefetching all limited-length protected streams.

Prototype

void __protected_stream_go (void);

Usage

Valid only when **-qarch** is set to target POWER5 processors or higher.

__protected_stream_set

Purpose

Establishes a limited-length protected stream which fetches from either incremental (forward) or decremental (backward) memory addresses. The stream is protected from being replaced by any hardware detected streams.

Prototype

void __protected_stream_set (unsigned int *direction*, const void* *addr*, unsigned int *stream_ID*);

Parameters

direction
An integer with a value of 1 (forward) or 3 (backward).

addr
The beginning of the cache line.

stream_ID
An integer with a value 0-7 on POWER5 processors, a value 0 to 15 on POWER6 processors, and a value 0 to 11 on POWER7 and POWER8 processors.

Usage

Valid only when **-qarch** is set to target POWER5 processors or higher.

__protected_unlimited_stream_set

Purpose

Establishes an unlimited-length protected stream which fetches from either incremental (forward) or decremental (backward) memory addresses. The stream is protected from being replaced by any hardware detected streams.

Prototype

void _protected_unlimited_stream_set (unsigned int *direction*, const void* *addr*, unsigned int *ID*);

Parameters

direction
An integer with a value of 1 (forward) or 3 (backward).

addr
The beginning of the cache line.

stream_ID
An integer with a value 0-7 on POWER5 processors, a value 0 to 15 on POWER6 processors, and a value 0 to 11 on POWER7 and POWER8 processors.

Usage

Valid only when **-qarch** is set to target POWER5 processors or higher.

__protected_stream_stride

Purpose

Sets the word-offset of the first unit of the stream address_offset, and stride in word size for protected load or store stream with identifier stream_id

Prototype

> void__protected_stream_stride (unsigned int *address_offset*, unsigned int *stride*, unsigned int *stream_id*);

Parameters

address_offset
The address of the first unit of the prefetch variable.

stride
This is the distance in the number of words of two consecutive elements of the prefetch stream.

stream_id
An integer with a value 0 to 11.

Usage

Valid only when -qarch is set to target POWER7 processors or higher.

__protected_stream_stop

Purpose

Stops prefetching a protected stream.

Prototype

> void __protected_stream_stop (unsigned int *stream ID*);

Parameters

stream_id
An integer with a value 0-7 on POWER5 processors, a value 0 to 15 on POWER6 processors, and a value 0 to 11 on POWER7 and POWER8 processors.

Usage

Valid only when **-qarch** is set to target POWER5 processors or higher.

__protected_stream_stop_all

Purpose

Stops prefetching all protected streams.

Prototype

> void __protected_stream_stop_all (void);

Usage

Valid only when **-qarch** is set to target POWER5 processors or higher.

__protected_store_stream_set

Purpose

Establishes a limited--length protected store stream which fetches from either incremental (forward) or decremental (backward) memory addresses. The stream is protected from being replaced by any hardware detected streams.

Prototype

> void _protected_store_stream_set (unsigned int *direction*, const void* *addr*, unsigned int *stream_ID*);

Parameters

direction
: An integer with a value of 1 (forward) or 3 (backward).

addr
: The beginning of the cache line.

stream_ID
: An integer with a value 0 to 15 on POWER6 processors, and a value 0 to 11 on POWER7 and POWER8 processors.

Usage

Valid only when **-qarch** is set to target POWER6 processors or higher.

__protected_unlimited_store_stream_set

Purpose

Establishes an unlimited-length protected store stream which fetches from either incremental (forward) or decremental (backward) memory addresses. The stream is protected from being replaced by any hardware detected streams.

Prototype

> void _protected_unlimited_store_stream_set (unsigned int *direction*, const void* *addr*, unsigned int *stream_ID*);

Parameters

direction
: An integer with a value of 1 (forward) or 3 (backward).

addr
The beginning of the cache line.

stream_ID
An integer with a value 0 to 15 on POWER6 processors, and a value 0 to 11 on POWER7 and POWER8 processors.

Usage

Valid only when **-qarch** is set to target POWER6 processors or higher.

__transient_protected_stream_count_depth

Purpose

Sets the number of cache lines *unit_cnt* and the prefetch depth *prefetch_depth* for the limited length protected load or store stream with identifier *stream_id*. The term "transient" indicates that the time interval during which the program accesses the stream's memory is likely to be short, so the processor can remove it from the cache earlier.

Prototype

> void __transient_protected_stream_count_depth (unsigned int *unit_cnt*, unsigned int *prefetch_depth*, unsigned int *stream_id*);

Parameters

unit_cnt
The number of cache lines. Must be an integer with a value of 0 to 1023.

prefetch_depth
A relative, qualitative value which sets the steady-state *fetch-ahead* distance of the prefetches for a stream. The fetch-ahead distance is the number of lines being prefetched in advance of the line from which data is currently being loaded, or to which data is currently being stored. Valid values are as follows:

0 The default defined in the Data Stream Control Register.

1 None.

2 Shallowest.

3 Shallow.

4 Medium.

5 Deep.

6 Deeper.

7 Deepest.

stream_id
An integer with a value 0 to 11.

Usage

Valid only when **-qarch** is set to target POWER7 processors or higher.

__transient_unlimited_protected_stream_depth

Purpose

Sets the prefetch depth prefetch_depth for the unlimited length protected load or store stream with identifier *stream_id*. The stream is likely to be transient, that is, the time interval during which the program accesses the unit is likely to be short.

Prototype

> void __transient_unlimited_protected_stream_depth (unsigned int *prefetch_depth*, unsigned int *stream_id*);

Parameters

prefetch_depth
A relative, qualitative value which sets the steady-state *fetch-ahead* distance of the prefetches for a stream. The fetch-ahead distance is the number of lines being prefetched in advance of the line from which data is currently being loaded, or to which data is currently being stored. Valid values are as follows:

0 The default defined in the Data Stream Control Register.

1 None.

2 Shallowest.

3 Shallow.

4 Medium.

5 Deep.

6 Deeper.

7 Deepest.

stream_id
An integer with a value 0 to 11.

Usage

Valid only when **-qarch** is set to target POWER7 processors or higher.

__unlimited_protected_stream_depth

Purpose

Sets the prefetch depth *prefetch_depth* for the unlimited length protected load or store stream with identifier *stream_id*.

Prototype

> void __unlimited_protected_stream_depth (unsigned in *prefetch_depth*, unsigned int *stream_id*);

Parameter

prefetch_depth
A relative, qualitative value which sets the steady-state *fetch-ahead* distance of the prefetches for a stream. The fetch-ahead distance is the number of lines being prefetched in advance of the line from which data is currently being loaded, or to which data is currently being stored. Valid values are as follows:

0 The default defined in the Data Stream Control Register.

1 None.

2 Shallowest.

3 Shallow.

4 Medium.

5 Deep.

6 Deeper.

7 Deepest.

stream_id
An integer with a value 0 to 15 on POWER6 processors, and a value 0 to 11 on POWER7 and POWER8 processors.

Usage

Valid only when **-qarch** is set to target POWER6 processors or higher.

Cryptography built-in functions

Cryptography built-in functions are valid only when **-qarch** is set to target POWER8 processors.

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*.

__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.

bzero

Purpose

Sets the first *n* bytes of the byte area starting at *s* to zero.

Prototype

void bzero(void* *s*, size_t *n*);

Parameters

n The size of the data.

s The starting address in the byte area.

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, such as POWER7. You must specify an appropriate **-qarch** suboption, such as**-qarch=pwr7**, when you use these types.

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.

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 46. 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 47. 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 48. 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 49. 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
vector unsigned int	vector bool int	vector unsigned int
	vector unsigned int	vector bool int
		vector unsigned int

Result value

The value of each element of the result is the saturated sum of the corresponding elements of a and b.

vec_add_u128

Purpose

Adds unsigned quadword values.

The function operates on vectors as 128-bit unsigned integers.

This built-in function is valid only when `-qarch` is set to target POWER8 processors.

Syntax

```
d=vec_add_u128(a, b)
```

Result and argument types

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

Result value

Returns low 128 bits of `a + b`.

vec_addc_u128

Purpose

Gets the carry bit of the 128-bit addition of two quadword values.

The function operates on vectors as 128-bit unsigned integers.

This built-in function is valid only when `-qarch` is set to target POWER8 processors.

Syntax

```
d=vec_addc_u128(a, b)
```

Result and argument types

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

Result value

Returns the carry out of `a + b`.

vec_adde_u128

Purpose

Adds unsigned quadword values with carry bit from the previous operation.

The function operates on vectors as 128-bit unsigned integers.

This built-in function is valid only when `-qarch` is set to target POWER8 processors.

Syntax

```
d=vec_adde_u128(a, b, c)
```

Result and argument types

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

Result value

Returns low 128 bits of a + b + (c & 1).

vec_addec_u128

Purpose

Gets the carry bit of the 128-bit addition of two quadword values with carry bit from the previous operation.

The function operates on vectors as 128-bit unsigned integers.

This built-in function is valid only when -qarch is set to target POWER8 processors.

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 50. 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 51. 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 52. Result and argument types

d	a	b
int	vector bool char	vector signed char
		vector unsigned char
	vector signed char	vector bool char
		vector signed char
	vector unsigned char	vector bool char
		vector unsigned char
	vector bool short	vector signed short
		vector unsigned short
	vector signed short	vector bool short
		vector signed short
	vector unsigned short	vector bool short
		vector unsigned short
	vector bool int	vector signed int
		vector unsigned int
	vector signed int	vector bool int
		vector signed int
	vector unsigned int	vector bool int
		vector unsigned int
	vector bool long long	vector signed long long
		vector unsigned long long
	vector signed long long	vector bool long long
		vector signed long long
	vector unsigned long long	vector bool long long
		vector unsigned long long
	vector float	vector float
	vector double	vector double

Result value

The result is 1 if all elements of a are 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 53. 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 54. 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 55. Result and argument types

d	a	b
int	vector bool char	vector signed char
		vector unsigned char
	vector signed char	vector bool char
		vector signed char
	vector unsigned char	vector bool char
		vector unsigned char
	vector bool short	vector signed short
		vector unsigned short
	vector signed short	vector bool short
		vector signed short
	vector unsigned short	vector bool short
		vector unsigned short
	vector bool int	vector signed int
		vector unsigned int
	vector signed int	vector bool int
		vector signed int
	vector unsigned int	vector bool int
		vector unsigned int
	vector bool long long	vector signed long long
		vector unsigned long long
	vector signed long long	vector bool long long
		vector signed long long
	vector unsigned long long	vector bool long long
		vector unsigned long long
	vector float	vector float
	vector double	vector double

Result value

The result is 1 if 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 56. 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 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 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 58. Result and argument types

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

Result value

The result is 1 if each element of a is not 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 59. Result and argument types

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

Result value

The result is 1 if each element of a is not 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 60. 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 61. 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 62. 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 63. 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 63. 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 64. 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 64. 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 65. 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 66. Result and argument types

d	a	b
int	vector bool char	vector signed char
		vector unsigned char
	vector signed char	vector bool char
		vector signed char
	vector unsigned char	vector bool char
		vector unsigned char
	vector bool short	vector signed short
		vector unsigned short
	vector signed short	vector signed short
		vector bool short
	vector unsigned short	vector bool short
		vector unsigned short
	vector bool int	vector signed int
		vector unsigned int
	vector signed int	vector bool int
		vector signed int
	vector unsigned int	vector bool int
		vector unsigned int
	vector bool long long	vector signed long long
		vector unsigned long long
	vector signed long long	vector bool long long
		vector signed long long
	vector unsigned long long	vector bool long long
		vector unsigned long long
	vector float	vector float
	vector double	vector double

Result value

The result is 1 if any element of `a` is 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 67. Result and argument types

d	a	b
int	vector bool char	vector signed char
		vector unsigned char
	vector signed char	vector bool char
		vector signed char
	vector unsigned char	vector bool char
		vector unsigned char
	vector bool short	vector signed short
		vector unsigned short
	vector signed short	vector signed short
		vector bool short
	vector unsigned short	vector bool short
		vector unsigned short
	vector bool int	vector signed int
		vector unsigned int
	vector signed int	vector bool int
		vector signed int
	vector unsigned int	vector bool int
		vector unsigned int
	vector bool long long	vector signed long long
		vector unsigned long long
	vector signed long long	vector bool long long
		vector signed long long
	vector unsigned long long	vector bool long long
		vector unsigned long long
	vector float	vector float
	vector double	vector double

Result value

The result is 1 if any element of a is 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 68. 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 69. 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 70. 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 71. 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 72. Result and argument types

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

Result value

The result is 1 if any element of a is not 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 73. Result and argument types

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

Result value

The result is 1 if any element of a is not 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 74. 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 75. 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 76. 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 77. 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 78. 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.

This built-in function is valid only when -qarch is set to target POWER8 processors.

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 563.

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 79. 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 tables describe the types of the returned value and the function arguments.

Table 80. Result and argument types

d	a	b
vector bool char	vector bool char	vector bool char
	vector signed char	vector signed char
	vector unsigned char	vector unsigned char
vector bool short	vector bool short	vector bool short
	vector signed short	vector signed short
	vector unsigned short	vector unsigned short
vector bool int	vector bool int	vector bool int
	vector signed int	vector signed int
	vector unsigned int	vector unsigned int
	vector float	vector float
vector bool long long	vector bool long long	vector bool long long
	vector double	vector double

When you call this built-in function, the following types are valid only when `-qarch` is set to target POWER8 processors.

Table 81. Result and argument types supported only on POWER8 processors

d	a	b
vector bool long long	vector signed long long	vector signed long long
	vector unsigned long long	vector unsigned long long

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 tables describe the types of the returned value and the function arguments.

Table 82. Result and argument types

d	a	b
vector bool char	vector signed char	vector signed char
	vector unsigned char	vector unsigned char

Table 82. Result and argument types (continued)

d	a	b
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 double	vector double

When you call this built-in function, the following types are valid only when `-qarch` is set to target POWER8 processors.

Table 83. Result and argument types supported only on POWER8 processors

d	a	b
vector bool long long	vector signed long long	vector signed long long
	vector unsigned long long	vector unsigned long long

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 tables describe the types of the returned value and the function arguments.

Table 84. Result and argument types

d	a	b
vector bool char	vector signed char	vector signed char
	vector unsigned char	vector unsigned char
vector bool short	vector signed short	vector signed short
	vector unsigned short	vector unsigned short

Table 84. Result and argument types (continued)

d	a	b
vector bool int	vector signed int	vector signed int
	vector unsigned int	vector unsigned int
	vector float	vector float
vector bool long long	vector double	vector double

When you call this built-in function, the following types are valid only when `-qarch` is set to target POWER8 processors.

Table 85. Result and argument types supported only on POWER8 processors

d	a	b
vector bool long long	vector signed long long	vector signed long long
	vector unsigned long long	vector unsigned long long

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 tables describe the types of the returned value and the function arguments.

Table 86. 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 double	vector double

When you call this built-in function, the following types are valid only when `-qarch` is set to target POWER8 processors.

Table 87. Result and argument types supported only on POWER8 processors

d	a	b
vector bool long long	vector signed long long	vector signed long long
	vector unsigned long long	vector unsigned long long

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 tables describe the types of the returned value and the function arguments.

Table 88. 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 double	vector double

When you call this built-in function, the following types are valid only when `-qarch` is set to target POWER8 processors.

Table 89. Result and argument types supported only on POWER8 processors

d	a	b
vector bool long long	vector signed long long	vector signed long long
	vector unsigned long long	vector unsigned long long

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.

This built-in function is valid only when -qarch is set to target POWER8 processors.

Syntax

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

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
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.

This built-in function is valid only when -qarch is set to target POWER7 processors or higher.

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 91. 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 92. 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 93. 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 94. 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 95. 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 96. 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 97. 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 98. Result and argument types

d	a
vector float	vector double
vector double	vector float

Result value

When this function converts from vector float to vector double, it converts the types of elements 0 and 2 in the vector.

When this function converts from vector double to vector float, the types of element 1 and 3 in the result vector are undefined.

vec_div

Purpose

Divides the elements in vector `a` by the corresponding elements in vector `b` and then assigns the result to corresponding elements in the result vector.

This function emulates the operation on integer vectors. This built-in function is valid only when `-qarch` is set to target POWER7 processors or higher.

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 99. 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 100. Types of the function arguments

a	b	c[1]
const signed char *	any integral type	unsigned int
const signed short *		
const signed int *		
const float *		

Note:

1. `c` must be an unsigned literal with a value in the range 0 - 3 inclusive.

vec_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 101. Types of the function arguments

a	b	c[1]
const signed char *	any integral type	unsigned int
const signed short *		
const signed int *		
const float *		

Note:

1. `c` must be an unsigned literal with a value in the range 0 - 3 inclusive.

vec_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 102. Types of the function arguments

a	b	c[1]
const signed char *	any integral type	unsigned int
const signed short *		
const signed int *		
const float *		

Note:

1. `c` must be an unsigned literal with a value in the range 0 - 3 inclusive.

vec_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 103. Types of the function arguments

a	b	c[1]
const signed char *	any integral type	unsigned int
const signed short *		
const signed int *		
const float *		

Note:

1. `c` must be an unsigned literal with a value in the range 0 - 3 inclusive.

vec_eqv

Purpose

Performs a bitwise equivalence operation on the input vectors.

This built-in function is valid only when `-qarch` is set to target POWER8 processors.

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 104. 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 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

Table 104. Types of the returned value and function arguments (continued)

d	a	b
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 105. 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 562.

vec_gbb

Purpose

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

This built-in function is valid only when `-qarch` is set to target POWER8 processors.

Syntax

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

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
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 107. 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 tables describe the types of the returned value and the function arguments.

Table 108. Data type of function returned value and arguments (in 32-bit mode)

d	a	b
vector float vector float	int	const vector float *
		const float *
vector signed int		const vector signed int *
		const signed int *
vector unsigned int		const vector unsigned int *
		const unsigned int *
vector signed short		const vector signed short *
		const signed short *
vector unsigned short		const vector unsigned short *
		const unsigned short *
vector signed char		const vector signed char *
		const signed char*
vector unsigned char		const vector unsigned char *
		const unsigned char *
vector bool char		const vector bool char *
vector bool int		const vector bool int *
vector bool short		const vector bool short *
vector pixel		const vector pixel *

Table 109. Data type of function returned value and arguments (in 64-bit mode)

d	a	b
vector unsigned int	int	const unsigned long*
vector signed int		const signed long*

Table 109. Data type of function returned value and arguments (in 64-bit mode) (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 110. 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 111. 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 tables describe the types of the returned value and the function arguments.

Table 112. Data type of function returned value and arguments (in 32-bit mode)

d	a	b
vector unsigned char	int	unsigned char*
		signed char*
		unsigned short*
		short*
		unsigned int*
		int*
		float*

Table 113. Data type of function returned value and arguments (in 64-bit mode)

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 tables describe the types of the returned value and the function arguments.

Table 114. Data type of function returned value and arguments (in 32-bit mode)

a	a	b
vector unsigned char	int	unsigned char*
		signed char*
		unsigned short*
		short*
		unsigned int*
		int*
		float*

Table 115. Data type of function returned value and arguments (in 64-bit mode)

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 116. 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 tables describe the types of the returned value and the function arguments.

Table 117. 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

When you call this built-in function, the following types are valid only when `-qarch` is set to target POWER8 processors.

Table 118. Result and argument types supported only on POWER8 processors

d	a	b
The same type as argument `a`	vector signed long long	The same type as argument `a`
	vector unsigned 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 119. 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 538

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 120. 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:

"vec_mergel"

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 121. 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:

"vec_mergeh" on page 536

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 122. 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 535

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 tables describe the types of the returned value and the function arguments.

Table 123. 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

Table 123. 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 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

When you call this built-in function, the following types are valid only when `-qarch` is set to target POWER8 processors.

Table 124. Result and argument types supported only on POWER8 processors

d	a	b
The same type as argument `a`	vector signed long long	The same type as argument `a`
	vector unsigned 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 125. 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.

This built-in function is valid only when `-qarch` is set to target POWER7 processors or higher.

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 126. 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 127. 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 4`n` to 4`n`+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 2`n` to 2`n`+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 128. Types of the returned value and function arguments

d	**a**	**b**	**c**
vector signed int	vector signed short	vector signed short	vector signed int
vector unsigned int	vector unsigned short	vector unsigned short	vector unsigned int

Result value

For each element n of the result vector, the value is obtained in the following way: multiply element p of a by element p of b, where p is from 2n to 2n+1; and then add the sum of these products to element n of c. All additions are performed by using 32-bit saturated arithmetic.

vec_mtvscr

Purpose

Copies the given value into the Vector Status and Control Register.

The low-order 32 bits of a are copied into the VSCR.

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.

This built-in function is valid only when `-qarch` is set to target POWER7 processors or higher.

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 129. 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 130. 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

Result value

Assume that the elements of each vector are numbered beginning with 0. For each element n of the result vector, the value is the product of the value of element 2n of a and the value of element 2n of b.

vec_mulo

Purpose

Returns a vector containing the results of multiplying every second set of corresponding elements of the given vectors, beginning with the second element.

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 131. 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

Result value

Assume that the elements of each vector are numbered beginning with 0. For each element n of the result vector, the value is the product of the value of element 2n+1 of a and the value of element 2n+1 of b.

vec_nabs

Purpose

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

This built-in function is valid only when `-qarch` is set to target POWER7 processors or higher.

Syntax

```
d=vec_nabs(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

This function computes the absolute value of each element in the given vector and then assigns the negated value of the result to the corresponding elements in the result vector.

vec_nand

Purpose

Performs a bitwise negated-and operation on the input vectors.

This built-in function is valid only when **-qarch** is set to target POWER8 processors.

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 133. Types of the returned value and function arguments

d	a	b
vector signed char	vector signed char	vector signed char
		vector bool char
vector unsigned char	vector unsigned char	vector unsigned char
		vector bool char

Table 133. Types of the returned value and function arguments (continued)

d	a	b
vector signed char	vector bool char	vector signed char
vector unsigned char		vector unsigned char
vector bool char		vector bool char
vector signed short	vector signed short	vector signed short
		vector bool short
vector unsigned short	vector unsigned short	vector unsigned short
		vector bool short
vector signed short	vector bool short	vector signed short
vector unsigned short		vector unsigned short
vector bool short		vector bool short
vector signed int	vector signed int	vector signed int
		vector bool int
vector unsigned int	vector unsigned int	vector unsigned int
		vector bool int
vector signed int	vector bool int	vector signed int
vector unsigned int		vector unsigned int
vector bool int		vector bool int
vector 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_neg

Purpose

Returns a vector containing the negated value of the corresponding elements in the given vector.

Note: For `vector signed long long`, this function emulates the operation. This built-in function is valid only when `-qarch` is set to target POWER7 processors or higher.

Syntax

```
d=vec_neg(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**
The same type as argument a	vector signed char
	vector signed short
	vector signed int
	vector signed long long
	vector float
	vector double

Result value

This function multiplies the value of each element in the given vector by -1.0 and then assigns the result to the corresponding elements in the result vector.

vec_nmadd

Purpose

Returns a vector containing the results of performing a negative multiply-add operation on the given vectors.

This built-in function is valid only when `-qarch` is set to target POWER7 processors or higher.

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 135. 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 136. 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 137. 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 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 138. 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.

This built-in function is valid only when **-qarch** is set to target POWER8 processors.

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 139. 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

Table 139. Types of the returned value and function arguments (continued)

d	a	b
vector unsigned long long	vector unsigned long long	vector unsigned long long
		vector bool long long
vector signed long long	vector bool long long	vector signed long long
vector unsigned long long		vector unsigned long long
vector bool long long		vector bool long long
vector double		vector double
vector float	vector float	vector bool int
		vector float
vector double	vector double	vector bool long long
		vector double

Result value

Each bit of the result is set to the result of the bitwise operation `(a | ~b)` of the corresponding bits of `a` and `b`. For 0 <= `i` < 128, bit `i` of the result is set to 1 only if the `i`th bit of `a` is 1 or the `i`th 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 tables describe the types of the returned value and the function arguments.

Table 140. 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

When you call this built-in function, the following types are valid only when `-qarch` is set to target POWER8 processors.

Table 141. Result and argument types supported only on POWER8 processors

d	a	b
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.

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 142. 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 tables describe the types of the returned value and the function arguments.

Table 143. 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

Table 143. Result and argument types (continued)

d	a	b
vector signed short	vector signed int	vector signed int
vector unsigned short	vector unsigned int	vector unsigned int

When you call this built-in function, the following types are valid only when `-qarch` is set to target POWER8 processors.

Table 144. Result and argument types supported only on POWER8 processors

d	a	b
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 tables describe the types of the returned value and the function arguments.

Table 145. 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

When you call this built-in function, the following types are valid only when `-qarch` is set to target POWER8 processors.

Table 146. Result and argument types supported only on POWER8 processors

d	a	b
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 147. 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		

Result value

Each byte of the result is selected by using the least significant five bits of the corresponding byte of c as an index into the concatenated bytes of a and b.

vec_permi

Purpose

Returns a vector by permuting and combining the two eight-byte-long vector elements in a and b based on the value of c.

Syntax

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

Result and argument types

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

Table 148. Result and argument types

d	a	b	c
vector bool long long	vector bool long long	vector bool long long	0–3
vector signed long long	vector signed long long	vector signed long long	
vector unsigned long long	vector unsigned long long	vector unsigned long long	
vector double	vector double	vector double	

Result value

If we use `a[0]` and `a[1]` to represent the first and second eight-byte-long elements in `a`, and use `b[0]` and `b[1]` for elements in `b`, then this function determines the elements in the result vector based on the binary value of `c`. This is illustrated as follows:

- 00 - `a[0], b[0]`
- 01 - `a[0]`, `b[1]`
- 10 - `a[1]`, `b[0]`
- 11 - `a[1]`, `b[1]`

vec_popcnt

Purpose

Computes the population count (number of set bits) in each element of the input.

This built-in function is valid only when `-qarch` is set to target POWER8 processors.

Syntax

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

Result and argument types

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

Table 149. Result and argument types

d	a
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

Table 149. Result and argument types (continued)

d	a
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 150. 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 151. 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_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 152. 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 153. 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_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 tables describe 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	The same type as argument a
	vector unsigned char	
	vector signed short	
	vector unsigned short	
	vector signed int	
	vector unsigned int	

When you call this built-in function, the following types are valid only when `-qarch` is set to target POWER8 processors.

Table 155. Result and argument types supported only on POWER8 processors

d	a	b
The same type as argument a	vector signed long long	The same type as argument a
	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 156. Result and argument types

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

Result value

Each element of the result contains the value of the corresponding element of `a`, rounded to the nearest representable floating-point integer, using IEEE round-to-nearest rounding.

vec_roundc

Purpose

Returns a vector by rounding every single-precision or double-precision floating-point element in the given vector to integer.

This built-in function is valid only when `-qarch` is set to target POWER7 processors or higher.

Syntax

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

Result and argument types

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

Table 157. Result and argument types

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

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 158. Result and argument types

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

Related reference:
"vec_floor" on page 526

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 159. Result and argument types

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

Related reference:
"vec_ceil" on page 510

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 160. 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 586

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 161. 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_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 162. 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_sl

Purpose

Performs a left shift for each element of a vector.

Syntax

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

Result and argument types

The following tables describe the types of the returned value and the function arguments.

Table 163. 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	

When you call this built-in function, the following types are valid only when `-qarch` is set to target POWER8 processors.

Table 164. Result and argument types supported only on POWER8 processors

d	a	b
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 165. 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 166. 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 167. 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 168. 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 169. 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 170. 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 171. 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 172. 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 173. 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 174. 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 175. 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 176. 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.

This built-in function is valid only when -qarch is set to target POWER7 processors or higher.

Syntax

```
d=vec_sqrt(a)
```

Result and argument types

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

Table 177. Result and argument types

d	a
vector 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 tables describe the types of the returned value and the function arguments.

Table 178. 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

When you call this built-in function, the following types are valid only when `-qarch` is set to target POWER8 processors.

Table 179. Result and argument types supported only on POWER8 processors

d	a	b
vector signed long long	vector signed long long	vector unsigned 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 tables describe the types of the returned value and the function arguments.

Table 180. 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	

When you call this built-in function, the following types are valid only when `-qarch` is set to target POWER8 processors.

Table 181. Result and argument types supported only on POWER8 processors

d	a	b
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 182. 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 183. 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 tables describe the types of the function arguments.

Table 184. Data type of function returned value and arguments (in 32-bit mode)

a	**b**	**c**
vector unsigned char	int	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*

Table 185. Data type of function returned value and arguments (in 64-bit mode)

a	**b**	**c**
vector unsigned int	int	unsigned long*
vector signed int		signed long*

Table 185. Data type of function returned value and arguments (in 64-bit mode) (continued)

a	b	c
vector unsigned char	long	vector unsigned char*
		unsigned char*
vector signed char		vector signed char*
		signed char*
vector bool char		vector bool char*
		unsigned char*
		signed char*
vector unsigned short		vector unsigned short*
		unsigned short*
vector signed short		vector signed short*
		signed short*
vector bool short		vector bool short*
		unsigned short*
		short*
vector pixel		vector pixel*
		unsigned short*
		short*
vector unsigned int		vector unsigned int*
		unsigned int*
vector signed int		vector signed int*
		signed int*
vector bool int		vector bool int*
		unsigned int*
		int*
vector float		vector float*
		float*

vec_ste

Purpose

Stores a vector element into memory at the given address.

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 186. 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 187. 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 188. Result and argument types

d	a	b
The same type as argument a	vector signed char	The same type as argument a
	vector unsigned char	
	vector signed short	
	vector unsigned short	
	vector signed int	
	vector unsigned int	
	vector signed long long	
	vector unsigned long long	
	vector float	
	vector double	

Result value

The value of each element of the result is the result of subtracting the value of the corresponding element of `b` from the value of the corresponding element of `a`. The arithmetic is modular for integer vectors.

vec_sub_u128

Purpose

Subtracts unsigned quadword values.

The function operates on vectors as 128-bit unsigned integers.

This built-in function is valid only when `-qarch` is set to target POWER8 processors.

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.

This built-in function is valid only when -qarch is set to target POWER8 processors.

Syntax

```
d=vec_subc_u128(a, b)
```

Result and argument types

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

Result value

Returns the carry out of a - b.

vec_sube_u128

Purpose

Subtracts unsigned quadword values with carry bit from previous operation.

The function operates on vectors as 128-bit unsigned integers.

This built-in function is valid only when -qarch is set to target POWER8 processors.

Syntax

```
d=vec_sube_u128(a, b, c)
```

Result and argument types

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

Result value

Returns the low 128 bits of a - b - (c & 1).

vec_subec_u128

Purpose

Gets the carry bit of the 128-bit subtraction of two quadword values with carry bit from the previous operation.

The function operates on vectors as 128-bit unsigned integers.

This built-in function is valid only when -qarch is set to target POWER8 processors.

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 189. 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 190. 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 563.

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 tables describe the types of the returned value and the function arguments.

Table 191. Result and argument types

d	a
vector signed short	vector signed char
vector signed int	vector signed short

When you call this built-in function, the following types are valid only when `-qarch` is set to target POWER8 processors.

Table 192. Result and argument types supported only on POWER8 processors

d	a
vector signed long long	vector signed int
vector bool long long	vector bool int

Result value

The value of each element of the result is the value of the corresponding element of the most significant half of `a`.

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 tables describe the types of the returned value and the function arguments.

Table 193. Result and argument types

d	a
vector signed short	vector signed char
vector signed int	vector signed short

When you call this built-in function, the following types are valid only when `-qarch` is set to target POWER8 processors.

Table 194. Result and argument types supported only on POWER8 processors

d	a
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`.

vec_xl

Purpose

Loads a 16-byte vector from the memory address specified by the displacement `a` and the pointer `b`.

This built-in function is valid only when `-qarch` is set to target POWER7 processors or higher.

Syntax

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

Result and argument types

The following tables describe the types of the function returned value and the function arguments in different bit modes.

Table 195. Data type of function returned value and arguments (in 32-bit mode)

d	a	b
vector signed char	int	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 *

Table 196. Data type of function returned value and arguments (in 64-bit mode)

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

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 different on little endian systems.

vec_xl_be

Purpose

Loads a 16-byte vector from the memory address specified by the displacement a and the pointer b.

This built-in function is valid only when -qarch is set to target POWER7 processors or higher.

Syntax

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

Result and argument types

The following tables describe the types of the function returned value and the function arguments in different bit modes.

Table 197. Data type of function returned value and arguments (in 32-bit mode)

d	a	b
vector signed char	int	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 *

Table 198. Data type of function returned value and arguments (in 64-bit mode)

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

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, even when the function is called on little endian systems.

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`.

This built-in function is valid only when `-qarch` is set to target POWER7 processors or higher.

Syntax

```
d=vec_xld2(a, b)
```

Result and argument types

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

Note: The type for operand `a` in the following table is: `int` in 32-bit mode, and `long` in 64-bit mode.

Table 199. Result and argument types

d	a	b
vector signed char	int	signed char *
	long	
vector unsigned char	int	unsigned char *
	long	

Table 199. Result and argument types (continued)

d	a	b
vector signed short	int	signed short *
	long	
vector unsigned short	int	unsigned short *
	long	
vector signed int	int	signed int *
	long	
vector unsigned int	int	unsigned int *
	long	
vector signed long long	int	signed long long *
	long	
vector unsigned long long	int	unsigned long long *
	long	
vector float	int	float *
	long	
vector double	int	double *
	long	

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_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.

This built-in function is valid only when `-qarch` is set to target POWER7 processors or higher.

Syntax

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

Result and argument types

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

Note: The type for operand `a` in the following table is: `int` in 32-bit mode, and `long` in 64-bit mode.

Table 200. Result and argument types

d	a	b
vector signed long long	int	signed long long *
	long	
vector unsigned long long	int	unsigned long long *
	long	
vector double	int	double *
	long	

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`.

This built-in function is valid only when `-qarch` is set to target POWER7 processors or higher.

Syntax

```
d=vec_xlw4(a, b)
```

Result and argument types

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

Note: The type for operand `a` in the following table is: `int` in 32-bit mode, and `long` in 64-bit mode.

Table 201. Result and argument types

d	a	b
vector signed char	int	signed char *
	long	
vector unsigned char	int	unsigned char *
	long	
vector signed short	int	signed short *
	long	
vector unsigned short	int	unsigned short *
	long	
vector signed int	int	signed int *
	long	

Table 201. Result and argument types (continued)

d	a	b
vector unsigned int	int	unsigned int *
	long	
vector float	int	float *
	long	

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_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 202. Result and argument types

d	a	b
vector bool char	vector bool char	vector bool char
vector signed char	vector bool char	vector signed char
	vector signed char	vector signed char
		vector bool char
vector unsigned char	vector bool char	vector unsigned char
	vector unsigned char	vector unsigned char
		vector bool char
vector bool short	vector bool short	vector vector bool short
vector signed short	vector bool short	vector signed short
	vector signed short	vector signed short
		vector bool short
vector unsigned short	vector bool short	vector unsigned short
	vector unsigned short	vector unsigned short
		vector bool short
vector bool int	vector bool int	vector bool int
vector signed int	vector bool int	vector signed int
	vector signed int	vector signed int
		vector bool int

Table 202. 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 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.

The order of vector elements stored to the effective address might be different on little-endian systems.

This built-in function is valid only when **-qarch** is set to target POWER7 processors or higher.

Syntax

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

Result and argument types

The following tables describe the types of the function returned value and the function arguments in different bit modes.

The element order of in argument `a` is different on little-endian systems.

Table 203. Data type of function returned value and arguments (in 32-bit mode)

d	a	b	c
void	vector signed char	int	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 *

Table 204. Data type of function returned value and arguments (in 64-bit mode)

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 *

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.

This built-in function is valid only when **-qarch** is set to target POWER7 processors or higher.

Syntax

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

Result and argument types

The following tables describe the types of the function returned value and the function arguments in different bit modes.

Table 205. Data type of function returned value and arguments (in 32-bit mode)

d	**a**	**b**	**c**
void	vector signed char	int	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 *

Table 206. Data type of function returned value and arguments (in 64-bit mode)

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 *

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`.

This built-in function is valid only when `-qarch` is set to target POWER7 processors or higher.

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.

Note: The type for operand a in the following table is: int in 32-bit mode, and long in 64-bit mode.

Table 207. Result and argument types

d	a	b	c
void	vector signed char	int	signed char *
		long	
	vector unsigned char	int	unsigned char *
		long	
	vector signed short	int	signed short *
		long	
	vector unsigned short	int	unsigned short *
		long	
	vector signed int	int	signed int *
		long	
	vector unsigned int	int	unsigned int *
		long	
	vector signed long long	int	signed long long *
		long	
	vector unsigned long long	int	unsigned long long *
		long	
	vector float	int	float *
		long	
	vector double	int	double *
		long	
	vector pixel	int	signed short *
			unsigned short *
		long	signed short *
			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.

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.

This built-in function is valid only when `-qarch` is set to target POWER7 processors or higher.

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.

Note: The type for operand `b` in the following table is: `int` in 32-bit mode, and `long` in 64-bit mode.

Table 208. Result and argument types

d	a	b	c
void	vector signed char	int	signed char *
		long	
	vector unsigned char	int	unsigned char *
		long	
	vector signed short	int	signed short *
		long	
	vector unsigned short	int	unsigned short *
		long	
	vector signed int	int	signed int *
		long	
	vector unsigned int	int	unsigned int *
		long	
	vector float	int	float *
		long	
	vector pixel	int	signed short *
			unsigned short *
		long	signed short *
			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.

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 209. Supported integral data types

`signed char`	`unsigned char`
`short int`	`unsigned short int`
`int`	`unsigned int`
`long int`	`unsigned long int`
`long long int` **1**	`unsigned long long int` **1**
	`_Bool`

1 **Restriction:** This type is supported only on 64-bit platforms.

Table 210. 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 ignores these optional parameters and protects all globally accessible variables.

The GCC atomic memory access built-in functions are grouped into the following categories.

Atomic lock, release, and synchronize functions

__sync_lock_test_and_set

Purpose

This function atomically assigns the value of *__v* to the variable that *__p* points to.

An acquire memory barrier is created when this function is invoked.

Prototype

T __sync_lock_test_and_set (*T** *__p*, *U* *__v*, ...);

Parameters

__p
The pointer of the variable that is to be set.

__v
The value to set to the variable that *__p* points to.

Return value

The function returns the initial value of the variable that *__p* points to.

__sync_lock_release

Purpose

This function releases the lock acquired by the `__sync_lock_test_and_set` function, and assigns the value of zero to the variable that *__p* points to.

A release memory barrier is created when this function is invoked.

Prototype

void __sync_lock_release (*T** *__p*, ...);

Parameters

__p
: The pointer of the variable that is to be set.

__sync_synchronize

Purpose

This function synchronizes data in all threads.

A full memory barrier is created when this function is invoked.

Prototype

void __sync_synchronize ();

Atomic fetch and operation functions

__sync_fetch_and_and

Purpose

This function performs an atomic bitwise AND operation on the variable *__v* with the variable that *__p* points to. The result is stored in the address that is specified by *__p*.

A full memory barrier is created when this function is invoked.

Prototype

T __sync_fetch_and_and (*T** *__p*, *U* *__v*, ...);

Parameters

__p
: The pointer of a variable on which the bitwise AND operation is to be performed. The value of this variable is to be changed to the result of the operation.

__v
: The variable with which the bitwise AND operation is to be performed.

Return value

The function returns the initial value of the variable that *__p* points to.

__sync_fetch_and_nand

Purpose

This function performs an atomic bitwise NAND operation on the variable *__v* with the variable that *__p* points to. The result is stored in the address that is specified by *__p*.

A full memory barrier is created when this function is invoked.

Prototype

T __sync_fetch_and_nand (*T* __p*, *U __v*, ...);

Parameters

__p
: The pointer of a variable on which the bitwise NAND operation is to be performed. The value of this variable is to be changed to the result of the operation.

__v
: The variable with which the bitwise NAND operation is to be performed.

Return value

The function returns the initial value of the variable that *__p* points to.

__sync_fetch_and_or

Purpose

This function performs an atomic bitwise inclusive OR operation on the variable *__v* with the variable that *__p* points to. The result is stored in the address that is specified by *__p*.

A full memory barrier is created when this function is invoked.

Prototype

T __sync_fetch_and_or (*T* __p*, *U __v*, ...);

Parameters

__p
: The pointer of a variable on which the bitwise inclusive OR operation is to be performed. The value of this variable is to be changed to the result of the operation.

__v
: The variable with which the bitwise inclusive OR operation is to be performed.

Return value

The function returns the initial value of the variable that *__p* points to.

__sync_fetch_and_xor

Purpose

This function performs an atomic bitwise exclusive OR operation on the variable *__v* with the variable that *__p* points to. The result is stored in the address that is specified by *__p*.

A full memory barrier is created when this function is invoked.

Prototype

> *T* __sync_fetch_and_xor (*T* * *__p*, *U* *__v*, ...);

Parameters

__p
: The pointer of a variable on which the bitwise exclusive OR operation is to be performed. The value of this variable is to be changed to the result of the operation.

__v
: The variable with which the bitwise exclusive OR operation is to be performed.

Return value

The function returns the initial value of the variable that *__p* points to.

__sync_fetch_and_add

Purpose

This function atomically adds the value of *__v* to the variable that *__p* points to. The result is stored in the address that is specified by *__p*.

A full memory barrier is created when this function is invoked.

Prototype

> *T* __sync_fetch_and_add (*T* * *__p*, *U* *__v*, ...);

Parameters

__p
: The pointer of a variable to which *__v* is to be added. The value of this variable is to be changed to the result of the add operation.

__v
: The variable whose value is to be added to the variable that *__p* points to.

Return value

The function returns the initial value of the variable that *__p* points to.

__sync_fetch_and_sub

Purpose

This function atomically subtracts the value of *__v* from the variable that *__p* points to. The result is stored in the address that is specified by *__p*.

A full memory barrier is created when this function is invoked.

Prototype

T __sync_fetch_and_sub (*T** *__p*, *U* *__v*, ...);

Parameters

__p
: The pointer of a variable from which *__v* is to be subtracted. The value of this variable is to be changed to the result of the sub operation.

__v
: The variable whose value is to be subtracted from the variable that *__p* points to.

Return value

The function returns the initial value of the variable that *__p* points to.

Atomic operation and fetch functions

__sync_and_and_fetch

Purpose

This function performs an atomic bitwise AND operation on the variable *__v* with the variable that *__p* points to. The result is stored in the address that is specified by *__p*.

A full memory barrier is created when this function is invoked.

Prototype

T __sync_and_and_fetch (*T** *__p*, *U* *__v*, ...);

Parameters

__p
: The pointer of a variable on which the bitwise AND operation is to be performed. The value of this variable is to be changed to the result of the operation.

__v
: The variable with which the bitwise AND operation is to be performed.

Return value

The function returns the new value of the variable that *__p* points to.

__sync_nand_and_fetch

Purpose

This function performs an atomic bitwise NAND operation on the variable *__v* with the variable that *__p* points to. The result is stored in the address that is specified by *__p*.

A full memory barrier is created when this function is invoked.

Prototype

T __sync_nand_and_fetch (*T** *__p*, *U* *__v*, ...);

Parameters

__p
: The pointer of a variable on which the bitwise NAND operation is to be performed. The value of this variable is to be changed to the result of the operation.

__v
: The variable with which the bitwise NAND operation is to be performed.

Return value

The function returns the new value of the variable that *__p* points to.

__sync_or_and_fetch

Purpose

This function performs an atomic bitwise inclusive OR operation on the variable *__v* with variable that *__p* points to. The result is stored in the address that is specified by *__p*.

A full memory barrier is created when this function is invoked.

Prototype

T __sync_or_and_fetch (*T** *__p*, *U* *__v*, ...);

Parameters

__p
: The pointer of a variable on which the bitwise inclusive OR operation is to be performed. The value of this variable is to be changed to the result of the operation.

__v
: The variable with which the bitwise inclusive OR operation is to be performed.

Return value

The function returns the new value of the variable that *__p* points to.

__sync_xor_and_fetch

Purpose

This function performs an atomic bitwise exclusive OR operation on the variable *__v* with the variable that *__p* points to. The result is stored in the address that is specified by *__p*.

A full memory barrier is created when this function is invoked.

Prototype

T __sync_xor_and_fetch (*T** *__p*, *U* *__v*, ...);

Parameters

__p
: The pointer of the variable on which the bitwise exclusive OR operation is to be performed. The value of this variable is to be changed to the result of the operation.

__v
: The variable with which the bitwise exclusive OR operation is to be performed.

Return value

The function returns the new value of the variable that *__p* points to.

__sync_add_and_fetch

Purpose

This function atomically adds the value of *__v* to the variable that *__p* points to. The result is stored in the address that is specified by *__p*.

A full memory barrier is created when this function is invoked.

Prototype

> *T* __sync_add_and_fetch (*T* __p*, *U __v*, ...);

Parameters

__p
: The pointer of a variable to which *__v* is to be added. The value of this variable is to be changed to the result of the add operation.

__v
: The variable whose value is to be added to the variable that *__p* points to.

Return value

The function returns the new value of the variable that *__p* points to.

__sync_sub_and_fetch

Purpose

This function atomically subtracts the value of *__v* from the variable that *__p* points to. The result is stored in the address that is specified by *__p*.

A full memory barrier is created when this function is invoked.

Prototype

> *T* __sync_sub_and_fetch (*T* __p*, *U __v*, ...);

Parameters

__p
: The pointer of a variable from which *__v* is to be subtracted. The value of this variable is to be changed to the result of the sub operation.

__v
The variable whose value is to be subtracted from the variable that *__p* points to.

Return value

The function returns the new value of the variable that *__p* points to.

Atomic compare and swap functions

__sync_val_compare_and_swap

Purpose

This function compares the value of *__compVal* to the value of the variable that *__p* points to. If they are equal, the value of *__exchVal* is stored in the address that is specified by *__p*; otherwise, no operation is performed.

A full memory barrier is created when this function is invoked.

Prototype

T __sync_val_compare_and_swap (*T* __p*, *U __compVal*, *V __exchVal*, ...);

Parameters

__p
The pointer to a variable whose value is to be compared with.

__compVal
The value to be compared with the value of the variable that *__p* points to.

__exchVal
The value to be stored in the address that *__p* points to.

Return value

The function returns the initial value of the variable that *__p* points to.

__sync_bool_compare_and_swap

Purpose

This function compares the value of *__compVal* with the value of the variable that *__p* points to. If they are equal, the value of *__exchVal* is stored in the address that is specified by *__p*; otherwise, no operation is performed.

A full memory barrier is created when this function is invoked.

Prototype

bool __sync_bool_compare_and_swap (*T* __p*, *U __compVal*, *V __exchVal*, ...);

Parameters

__p
The pointer to a variable whose value is to be compared with.

__compVal
The value to be compared with the value of the variable that *__p* points to.

__exchVal
The value to be stored in the address that *__p* points to.

Return value

If the value of *__compVal* and the value of the variable that *__p* points to are equal, the function returns `true`; otherwise, it returns `false`.

Miscellaneous built-in functions

Miscellaneous functions are grouped into the following categories:

- "Optimization-related functions"
- "Move to/from register functions" on page 608
- "Memory-related functions" on page 610

Optimization-related functions

__alignx

Purpose

Allows for optimizations such as automatic vectorization by informing the compiler that the data pointed to by *pointer* is aligned at a known compile-time offset.

Prototype

void __alignx (int *alignment*, const void* *pointer*);

Parameters

alignment
Must be a constant integer with a value greater than zero and of a power of two.

__builtin_expect

Purpose

Indicates that an expression is likely to evaluate to a specified value. The compiler may use this knowledge to direct optimizations.

Prototype

long __builtin_expect (long *expression*, long *value*);

Parameters

expression
Should be an integral-type expression.

value
Must be a constant literal.

Usage

If the *expression* does not actually evaluate at run time to the predicted value, performance may suffer. Therefore, this built-in function should be used with caution.

__fence

Purpose

Acts as a barrier to compiler optimizations that involve code motion, or reordering of machine instructions. Compiler optimizations will not move machine instructions past the location of the `__fence` call.

Prototype

void __fence (void);

Examples

This function is useful to guarantee the ordering of instructions in the object code generated by the compiler when optimization is enabled.

Move to/from register functions

__mftb

Purpose

Move from Time Base

In 32-bit compilation mode, returns the lower word of the time base register. In 64-bit mode, returns the entire doubleword of the time base register.

Prototype

unsigned long __mftb (void);

Usage

In 32-bit mode, this function can be used in conjunction with the`__mftbu` built-in function to read the entire time base register. In 64-bit mode, the entire doubleword of the time base register is returned, so separate use of `__mftbu` is unnecessary

It is recommended that you insert the `__fence` built-in function before and after the `__mftb` built-in function.

__mftbu

Purpose

Move from Time Base Upper

Returns the upper word of the time base register.

Prototype

unsigned int __mftbu (void);

Usage

In 32-bit mode you can use this function in conjunction with the `__mftb` built-in function to read the entire time base register

It is recommended that you insert the `__fence` built-in function before and after the `__mftbu` built-in function.

__mfmsr

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.

Related information

- "Register transfer functions" on page 447

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 256.

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 256

Built-in functions for parallel processing

Use these built-in functions to obtain information about the parallel environment:

- "IBM SMP built-in functions" on page 613
- Chapter 8, "OpenMP runtime functions for parallel processing," on page 621
- "Transactional memory built-in functions" on page 613

IBM SMP built-in functions

__parthds

Purpose

Returns the value of the **parthds** runtime option.

Prototype

```
int __parthds (void);
```

Return value

If the **parthds** option is not explicitly set, returns the default value set by the runtime library. If the **-qsmp** compiler option was not specified during program compilation, returns 1 regardless of runtime options selected.

__usrthds

Purpose

Returns the value of the **usrthds** runtime option.

Prototype

```
int __usrthds (void);
```

Return value

If the **usrthds** option is not explicitly set, or the **-qsmp** compiler option is not specified during program compilation, returns `0` regardless of runtime options selected.

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.

Notes:

- Transactional memory built-in functions are valid only when **-qarch** is set to target POWER8 processors.
- You must include the `htmxlintrin.h` file in the source code if you use any of the transactional memory built-in functions.

Transaction begin and end functions

__TM_begin:
Purpose

Marks the beginning of a transaction.

Prototype

long __TM_begin (void* const *TM_buff*);

Parameter

`TM_buff`
The address of a 16-byte transaction diagnostic block (TDB) that contains diagnostic information.

Usage

Upon a transaction failure (including a user abort), execution resumes from the point immediately following the `__TM_begin` that initiated the failed transaction as if the `__TM_begin` were unsuccessful. The diagnostic information is transferred from the TEXASR and TFIAR registers to *TM_buff*.

You can use the transaction inquiry functions to query the transaction status.

Return value

This function returns _HTM_TBEGIN_STARTED if successful; otherwise, it returns a different value.

Related information

- "__TM_simple_begin"
- "Transaction inquiry functions" on page 616

__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 614
- "Transaction inquiry functions" on page 616

Transaction abort functions

__TM_abort:
Purpose

Aborts a transaction with failure code 0.

Prototype

void __TM_abort ();

Related information

- "__TM_named_abort"

__TM_named_abort:
Purpose

Aborts a transaction with the specified failure code.

Prototype

void __TM_named_abort (unsigned char const *code*);

Parameter

code
The specified failure code. It is a literal that is in the range of 0 - 255.

Related information

- "__TM_abort" on page 615

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 618

__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.

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 25

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.

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"

Related information:

"OMP_SCHEDULE" on page 36

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"

Related information:

"OMP_SCHEDULE" on page 36

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_num_threads

Purpose

Returns the number of threads currently in the team executing the parallel region from which it is called.

Prototype

```
int omp_get_num_threads (void);
```

omp_set_num_threads

Purpose

Overrides the setting of the *OMP_NUM_THREADS* environment variable, and specifies the number of threads to use for a subsequent parallel region by setting the first value of *num_list* for *OMP_NUM_THREADS*.

Prototype

```
void omp_set_num_threads (int num_threads);
```

Parameters

num_threads
Must be a positive integer.

Usage

If the `num_threads` clause is present, then for the parallel region it is applied to, it supersedes the number of threads requested by this function or the *OMP_NUM_THREADS* environment variable. Subsequent parallel regions are not affected by it.

omp_get_max_threads

Purpose

Returns the first value of *num_list* for the *OMP_NUM_THREADS* environment variable. This value is the maximum number of threads that can be used to form a new team if a parallel region without a **num_threads** clause is encountered.

Prototype

```
int omp_get_max_threads (void);
```

omp_get_thread_num

Purpose

Returns the thread number, within its team, of the thread executing the function.

Prototype

```
int omp_get_thread_num (void);
```

Return value

The thread number lies between 0 and **omp_get_num_threads()**-1 inclusive. The master thread of the team is thread 0.

omp_get_num_procs

Purpose

Returns the maximum number of processors that could be assigned to the program.

Prototype

```
int omp_get_num_procs (void);
```

omp_in_final

Purpose

Returns a nonzero integer value if the function is called in a final task region; otherwise, it returns 0.

Prototype

```
int omp_in_final(void);
```

omp_in_parallel

Purpose

Returns non-zero if it is called within the dynamic extent of a parallel region executing in parallel; otherwise, returns 0.

Prototype

int omp_in_parallel (void);

omp_set_dynamic

Purpose

Enables or disables dynamic adjustment of the number of threads available for execution of parallel regions.

Prototype

void omp_set_dynamic (int *dynamic_threads*);

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 for AIX.

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who want to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Intellectual Property Dept. for Rational Software
IBM Corporation
5 Technology Park Drive
Westford, MA 01886
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided “AS IS”, without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. 1998, 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.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Index

Special characters

- -qassert compiler option 108
- -qdbgfmt compiler option 129
- -qfdpr compiler option 144
- -qflttrap compiler option 151
- -qfunctrace 158
- -qhelp compiler option 169
- -qlibmpi 215
- -qlistfmt compiler option 218
- -qoptdebug compiler option 239
- -qreport compiler option 263
- -qsaveopt compiler option 272
- -qskipsrc compiler option 280
- -qsmp compiler option 281
- -qstackprotect compiler option 291
- -qversion compiler option 321
- #pragma nofunctrace 158, 361

A

- alias 96
  - -qalias compiler option 96
  - pragma disjoint 345
- alignment 98
  - -qalign compiler option 98
  - pragma align 98
  - pragma pack 366
- alter program semantics 294
- appending macro definitions, preprocessed output 276
- architecture 9, 102
  - -q32 compiler option 94
  - -q64 compiler option 94
  - -qarch compiler option 102
  - -qcache compiler option 116
  - -qtune compiler option 310
  - architecture combination 312
  - macros 407
- arrays
  - padding 169

B

- backward 15
- Backward compatibility issues 15
- basic example, described xiii
- built-in functions 413
  - BCD 433
  - Binary-coded decimal 433
    - __bcd_invalid 435
    - __bcdadd 434
    - __bcdadd_ofl 434
    - __bcdcmpeq 435
    - __bcdcmpge 435
    - __bcdcmpgt 435
    - __bcdcmple 436
    - __bcdcmplt 436
    - __bcdsub 434
    - __bcdsub_ofl 435
    - vec_ldrmb 436
- built-in functions *(continued)*
  - Binary-coded decimal *(continued)*
    - vec_strmb 436
  - block-related 479
  - cache-related 464
  - cryptography 474
    - __vcipher 474
    - __vcipherlast 475
    - __vncipher 475
    - __vncipherlast 475
    - __vpermxor 477
    - __vpmsumb 478
    - __vpmsumd 478
    - __vpmsumh 479
    - __vpmsumw 479
    - __vsbox 476
    - __vshasigmad 476
    - __vshasigmaw 477
  - fixed-point 413
  - floating-point 422, 437
    - binary 422
    - decimal 437
  - for parallel processing 612
  - GCC atomic memory access 598
  - miscellaneous 607
  - synchronization and atomic 456
  - transactional memory 613

C

- cleanpdf command 250
- compatibility 15
  - compatibility
    - options for compatibility 90
- compiler options 5
  - architecture-specific 9
  - performance optimization 85
  - resolving conflicts 8
  - specifying compiler options 5
    - command line 5
    - configuration file 7
    - source files 7
  - summary of command line options 75
- compiler predefined macros 403
- configuration 38
  - custom configuration files 38
  - gxlc options 42
  - specifying compiler options 7
- configuration file 143
- control of transformations 294

D

- data types 101
  - -qaltivec compiler option 101
- debug optimized code 239
- Dynamic Probe Class Library
  - -qdpcl compiler option 134

E

- environment variables
  - compile-time and link-time 24
  - OpenMP
    - OMP_DYNAMIC 33
    - OMP_PROC_BIND 35
    - OMP_STACKSIZE 37
    - OMP_THREAD_LIMIT 37
    - OMP_WAIT_POLICY 38
  - runtime
    - XLSMPOPTS 26
  - scheduling algorithm environment variable 36
  - setting 23
  - XLSMPOPTS environment variable 25
- error checking and debugging 81
  - -g compiler option 160
  - -qcheck compiler option 119
  - -qheapdebug compiler option 167
  - -qlinedebug compiler option 216
- exception handling
  - for floating point 151

F

- fini 349
- floating-point
  - exceptions 151
- function trace 158

G

- GCC options 11
- gxlc utility 11

H

- high order transformation 169

I

- implicit timestamps 307
- init pragma 355
- inlining 189
- interprocedural analysis (IPA) 193
- invocations 1
  - compiler or components 1
  - preprocessor 12
  - selecting 1
  - syntax 2

L

- language level 206
- language standards 206
- large pages 211
- lib*.a library files 204

lib*.so library files 204
libraries
  redistributable 14
  XL C 14
linker 13
  -G compiler option 163
  invoking 13
linking 13
  -brtl compiler option 113
  -G compiler option 163
  options that control linking 89
  order of linking 14
listing 19, 275
  -qattr compiler option 108
  -qlist compiler option 217
  -qlistopt compiler option 221
  -qsource compiler option 286
  -qxref compiler option 330
  options that control listings and messages 84

M

macro definitions, preprocessed output 276
macros
  related to architecture 407
  related to compiler options 406
  related to language features 408
  related to the compiler 404
  related to the platform 405
maf suboption of -qfloat 297
mergepdf 250
mpi 215
MPI 215

N

nofunctrace 361

O

OMP_DISPLAY_ENV environment variable 31
OMP_DYNAMIC environment variable 33
OMP_MAX_ACTIVE_LEVELS 33
OMP_NESTED environment variable 33
OMP_NUM_THREADS environment variable 34
OMP_PROC_BIND environment variable 35
OMP_SCHEDULE environment variable 36
OMP_STACKSIZE environment variable 37
OMP_THREAD_LIMIT environment variable 37
OMP_WAIT_POLICY environment variable 38
OpenMP 31
OpenMP environment variables 31, 37, 38
optimization 85
  -O compiler option 236
  -qalias compiler option 96

optimization *(continued)*
  -qoptimize compiler option 236
  controlling, using option_override pragma 364
  loop optimization 85
    -qhot compiler option 169
    -qstrict_induction compiler option 299
  options for performance optimization 85

P

parallel processing 31
  built-in functions 612
  OpenMP environment variables 31
  parallel processing pragmas 380
  pragma directives 380
  setting parallel processing environment variables 25
performance 85
  -O compiler option 236
  -qalias compiler option 96
  -qoptimize compiler option 236
pragma nofunctrace 361
pragmas
  fini 349
  init 355
  unroll 375
procedure trace 158
profile-directed feedback (PDF) 247
  -qpdf1 compiler option 247
  -qpdf2 compiler option 247
profiling 243
  -qdpcl compiler option 134
  -qpdf1 compiler option 247
  -qpdf2 compiler option 247
  -qshowpdf compiler option 277

R

rrm suboption of -qfloat 297

S

shared objects 233
  -b compiler option 109
  -qmkshrobj 233
shared-memory parallelism (SMP) 26
  -qsmp compiler option 281
  environment variables 26
showpdf 250
SIGTRAP signal 151
skipsrc
  skipsrc 280
stackprotect
  stackprotect 291
static assertions
  -qlanglvl compiler option
    -qlanglvl=extc1x 206

T

target machine 102
transformations, control of 294
tuning 310
  -qarch compiler option 310
  -qtune compiler option 310

V

vector built-in functions
  vec_abs 481
  vec_abss 481
  vec_add 482
  vec_add_u128 484
  vec_addc 482
  vec_addc_u128 484
  vec_adde_u128 484
  vec_addec_u128 485
  vec_adds 483
  vec_all_in 489
  vec_and 496
  vec_andc 497
  vec_any_out 509
  vec_avg 509
  vec_bperm 510
  vec_ceil 510
  vec_cmpb 511
  vec_cmpeq 511
  vec_cmpgt 513
  vec_cmplt 515
  vec_cntlz 516
  vec_cpsgn 516
  vec_dss 521
  vec_dssall 521
  vec_dst 521
  vec_dstst 522
  vec_dststt 522
  vec_dstt 523
  vec_eqv 523
  vec_expte 525
  vec_extract 525
  vec_floor 526
  vec_gbb 526
  vec_insert 527
  vec_ld 528
  vec_lde 529
  vec_ldl 530
  vec_loge 531
  vec_lvsl 532
  vec_lvsr 532
  vec_madd 533
  vec_madds 534
  vec_mergee 535
  vec_mergeo 538
  vec_mfvscr 539
  vec_mladd 540
  vec_mradds 541
  vec_msum 542
  vec_msums 543
  vec_mtvscr 543
  vec_mul 544
  vec_mule 544
  vec_mulo 545
  vec_nabs 546
  vec_nand 546
  vec_neg 548
  vec_nor 549
  vec_orc 552
  vec_pack 553
  vec_packpx 554

vector built-in functions *(continued)*
  vec_packs 554
  vec_packsu 555
  vec_perm 556
  vec_revb 559
  vec_reve 560
  vec_rl 561
  vec_round 561
  vec_sl 565
  vec_sld 566
  vec_sldw 567
  vec_sll 568
  vec_slo 569
  vec_splat 570
  vec_splat_s16 571
  vec_splat_s32 572
  vec_splat_s8 571
  vec_splat_u16 573
  vec_splat_u32 573
  vec_splat_u8 572
  vec_splats 570
  vec_sr 574
  vec_sra 575
  vec_srl 576
  vec_sro 577
  vec_st 577
  vec_ste 579
  vec_stl 580
  vec_sub_u128 582
  vec_subc 583
  vec_subc_u128 583
  vec_sube_u128 583
  vec_subec_u128 584
  vec_subs 584
  vec_sum2s 585
  vec_sum4s 585
  vec_sums 586
  vec_trunc 586
  vec_unpackh 586
  vec_unpackl 587
vector data types 101
  -qaltivec compiler option 101
vector processing 278
  -qaltivec compiler option 101
visibility attributes 323
  pragma directive 349
VMX built-in functions
  vec_xl 588
  vec_xl_be 589
  vec_xst_be 595

X

XLSMPOPTS environment variable 26

Product Number: 5765-J06; 5725-C71

Printed in USA

SC27-4239-02