

IBM XL Fortran Advanced Edition V9.1 for Linux



User's Guide

IBM XL Fortran Advanced Edition V9.1 for Linux



User's Guide

Note!

Before using this information and the product it supports, be sure to read the general information under “Notices” on page 363.

Second Edition (March 2005)

- | This edition applies to Version 9.1.1 of IBM® XL Fortran Advanced Edition V9.1 for Linux™ (Program 5724-K76) 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. Changes from the previous edition of this book are identified by a vertical bar (|) in the left hand margin.

IBM welcomes your comments. You can send your comments electronically to the network ID listed below. Be sure to include your entire network address if you wish a reply.

- Internet: compinfo@ca.ibm.com

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 1990, 2005. All rights reserved.

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

Contents

Figures	vii
--------------------------	------------

What's New for XL Fortran	ix
--	-----------

Introduction	1
-------------------------------	----------

How to Use This Document	1
How to Read the Syntax Diagrams and Statements	2
Notes on the Examples in This Document	4
Notes on the Path Names in This Document.	4
Notes on the Terminology in This Document	4
Typographical Conventions	4
Related Documentation	4
XL Fortran and Operating System Publications	4
Standards Documents	5

Overview of XL Fortran Features	7
--	----------

Hardware and Operating-System Support	7
Language Support	7
Migration Support	8
Source-Code Conformance Checking	8
Highly Configurable Compiler	8
Diagnostic Listings	9
Symbolic Debugger Support	9
Program Optimization	9
Documentation and Online Help	9

Setting Up and Customizing XL Fortran 11
--

Where to Find Installation Instructions	11
Using the Compiler on a Network File System.	11
Correct Settings for Environment Variables	12
Environment Variable Basics.	12
Environment Variables for National Language Support	12
Setting Library Search Paths.	13
PDFDIR: Specifying the Directory for PDF Profile Information	14
TMPDIR: Specifying a Directory for Temporary Files	14
XLFSCRATCH_unit: Specifying Names for Scratch Files	14
XLFUNIT_unit: Specifying Names for Implicitly Connected Files	14
Customizing the Configuration File	14
Attributes	15
What a Configuration File Looks Like	17
Determining Which Level of XL Fortran Is Installed	20
Running Two Levels of XL Fortran	20

Editing, Compiling, Linking, and Running XL Fortran Programs	21
---	-----------

Editing XL Fortran Source Files.	21
Compiling XL Fortran Programs	21
Compiling Fortran 90 or Fortran 95 Programs	22
Compiling XL Fortran SMP Programs	23

Compilation Order for Fortran Programs	23
Canceling a Compilation	24
XL Fortran Input Files	24
XL Fortran Output Files	25
Scope and Precedence of Option Settings	26
Specifying Options on the Command Line	27
Specifying Options in the Source File.	27
Passing Command-Line Options to the "ld" or "as" Commands	28
Displaying Information inside Binary Files (strings)	29
Compiling for Specific Architectures	29
Passing Fortran Files through the C Preprocessor	29
cpp Directives for XL Fortran Programs	31
Passing Options to the C Preprocessor	31
Avoiding Preprocessing Problems	31
Linking XL Fortran Programs	32
Compiling and Linking in Separate Steps	32
Passing Options to the ld Command	32
Dynamic and Static Linking.	32
Avoiding Naming Conflicts during Linking.	33
Running XL Fortran Programs	33
Canceling Execution	34
Compiling and Executing on Different Systems	34
Run-Time Libraries for POSIX Pthreads Support	34
Selecting the Language for Run-Time Messages	34
Setting Run-Time Options	35
OpenMP Environment Variables	47
Other Environment Variables That Affect Run-Time Behavior	49
XL Fortran Run-Time Exceptions	49

XL Fortran Compiler-Option Reference 51

Summary of the XL Fortran Compiler Options.	51
Options That Control Input to the Compiler	52
Options That Specify the Locations of Output Files	54
Options for Performance Optimization	54
Options for Error Checking and Debugging	58
Options That Control Listings and Messages	61
Options for Compatibility	62
Options for Floating-Point Processing.	70
Options That Control Linking	70
Options That Control Other Compiler Operations	71
Options That Are Obsolete or Not Recommended	71
Detailed Descriptions of the XL Fortran Compiler Options.	73
-# Option	74
-1 Option	75
-B Option	76
-C Option	77
-c Option	78
-D Option	79
-d Option	80
-F Option	81

-g Option	82	-qmixed Option	167
-I Option	83	-qmoddir Option	168
-k Option	84	-qmodule Option	169
-L Option	85	-qnoprint Option	170
-l Option	86	-qnullterm Option	171
-N Option	87	-qobject Option	172
-O Option	88	-qonetrip Option	173
-o Option	90	-qoptimize Option	174
-p Option	91	-qpdf Option	175
-Q Option	92	-qphsinfo Option	179
-q32 Option	93	-qpic Option	181
-q64 Option	94	-qport Option	182
-qalias Option	95	-qposition Option	184
-qalign Option	98	-qprefetch Option	185
-qarch Option	100	-qqcount Option	186
-qassert Option	104	-qrealize Option	187
-qattr Option	105	-qrecur Option	189
-qautodbl Option	106	-qreport Option	190
-qbigdata Option	108	-qsaas Option	192
-qcache Option	109	-qsave Option	193
-qcclines Option	111	-qsaveopt Option	194
-qcheck Option	112	-qsclock Option	195
-qci Option	113	-qshowpdf Option	196
-qcompact Option	114	-qsigtrap Option	197
-qcr Option	115	-qsmallstack Option	198
-qctyplss Option	116	-qsmp Option	199
-qdbg Option	118	-qsource Option	204
-qddim Option	119	-qspillsize Option	205
-qdirective Option	120	-qstrict Option	206
-qdirectstorage Option	122	-qstrictieemod Option	207
-qdlines Option	123	-qstrict_induction Option	208
-qdpc Option	124	-qsuffix Option	209
-qenablevmx Option	125	-qsuppress Option	210
-qescape Option	126	-qswapomp Option	212
-qessl Option	127	-qhtable Option	214
-qextern Option	128	-qthreaded Option	215
-qextername Option	129	-qtune Option	216
-qfixed Option	131	-qundef Option	218
-qflag Option	132	-qunroll Option	219
-qfloat Option	133	-qunwind Option	220
-qfltrap Option	135	-qversion Option	221
-qfree Option	137	-qwarn64 Option	222
-qfullpath Option	138	-qxflag=dvz Option	223
-qhalt Option	139	-qxflag=oldtab Option	224
-qhot Option	140	-qxlf77 Option	225
-qieee Option	142	-qxlf90 Option	227
-qinit Option	143	-qxlines Option	229
-qinitauto Option	144	-qxref Option	231
-qintlog Option	146	-qzerosize Option	232
-qintsize Option	147	-S Option	233
-qipa Option	149	-t Option	234
-qkeepparm Option	155	-U Option	235
-qlanglvl Option	156	-u Option	236
-qlibansi Option	158	-v Option	237
-qlibposix Option	159	-V Option	238
-qlist Option	160	-W Option	239
-qlistopt Option	161	-w Option	241
-qlog4 Option	162	-y Option	242
-qmaxmem Option	163		
-qmbcs Option	165		
-qminimaltoc Option	166		

Using XL Fortran in a 64-Bit Environment. 243

Compiler Options for the 64-Bit Environment	243
-q32 Option	244
-q64 Option	245
-qwarn64 Option	247

XL Fortran Floating-Point Processing 249

IEEE Floating-Point Overview	249
Compiling for Strict IEEE Conformance	249
IEEE Single- and Double-Precision Values	250
IEEE Extended-Precision Values	250
Infinities and NaNs	250
Exception-Handling Model	251
Hardware-Specific Floating-Point Overview	252
Single- and Double-Precision Values	252
Extended-Precision Values	253
How XL Fortran Rounds Floating-Point Calculations	254
Selecting the Rounding Mode	254
Minimizing Rounding Errors	256
Minimizing Overall Rounding	256
Delaying Rounding until Run Time	256
Ensuring that the Rounding Mode is Consistent	256
Duplicating the Floating-Point Results of Other Systems	257
Maximizing Floating-Point Performance	257
Detecting and Trapping Floating-Point Exceptions	257
Compiler Features for Trapping Floating-Point Exceptions	258
Installing an Exception Handler	258
Controlling the Floating-Point Status and Control Register	260
xlf_fp_util Procedures	260
fpgets and fpsets Subroutines	261
Sample Programs for Exception Handling	262
Causing Exceptions for Particular Variables	263
Minimizing the Performance Impact of Floating-Point Exception Trapping	263

Optimizing XL Fortran Programs 265

The Philosophy of XL Fortran Optimizations	265
Summary of Compiler Options for Optimization	267
Choosing an Optimization Level	267
Optimization Level -O2	268
Optimization Level -O3	268
Getting the Most out of -O2 and -O3	269
The -O4 and -O5 Options	269
Optimizing for a Target Machine or Class of Machines	270
Getting the Most out of Target Machine Options	270
Optimizing Floating-Point Calculations	271
High-order Transformations (-qhot)	271
Getting the Most out of -qhot	272
Optimizing Loops and Array Language	272
Profile-directed Feedback (PDF)	275
Using Profile-directed Feedback (PDF)	275
Optimizing Conditional Branching	276
Interprocedural Analysis (-qipa)	276
Getting the Most from -qipa	277
Optimizing Subprogram Calls	277
Finding the Right Level of Inlining	278

Shared-memory Parallelism (-qsmp)	279
Getting the Most out of -qsmp	279
Other Program Behavior Options	280
Other Performance Options	280
Debugging Optimized Code	281
Different Results in Optimized Programs	282
Compiler-friendly Programming	282

Implementation Details of XL Fortran Input/Output 285

Implementation Details of File Formats	285
File Names	286
Preconnected and Implicitly Connected Files	286
File Positioning	287
I/O Redirection	288
How XLF I/O Interacts with Pipes, Special Files, and Links	288
Default Record Lengths	289
File Permissions	289
Selecting Error Messages and Recovery Actions	289
Flushing I/O Buffers	290
Choosing Locations and Names for Input/Output Files	290
Naming Files That Are Connected with No Explicit Name	290
Naming Scratch Files	291
Asynchronous I/O	292
Execution of an Asynchronous Data Transfer Operation	292
Usage	292
Performance	294
Compiler-Generated Temporary I/O Items	295
Error Handling	296
XL Fortran Thread-Safe I/O Library	296
Use of I/O Statements in Signal Handlers	299
Asynchronous Thread Cancellation	299

Interlanguage Calls 301

Conventions for XL Fortran External Names	301
Mixed-Language Input and Output	302
Mixing Fortran and C++	302
Making Calls to C Functions Work	304
Passing Data From One Language to Another	305
Passing Arguments between Languages	305
Passing Global Variables between Languages	306
Passing Character Types between Languages	306
Passing Arrays between Languages	307
Passing Pointers between Languages	308
Passing Arguments By Reference or By Value	309
Passing Complex Values to/from gcc	311
Returning Values from Fortran Functions	311
Arguments with the OPTIONAL Attribute	311
Arguments with the INTENT Attribute	311
Type Encoding and Checking	311
Assembler-Level Subroutine Linkage Conventions	312
The Stack	313
The Link Area and Minimum Stack Frame	315
The Input Parameter Area	316
The Register Save Area	316
The Local Stack Area	316

The Output Parameter Area	316
Linkage Convention for Argument Passing	317
Argument Passing Rules (by Value)	318
Order of Arguments in Argument List	320
Linkage Convention for Function Calls	320
Pointers to Functions	321
Function Values	321
The Stack Floor	322
Stack Overflow	322
Prolog and Epilog	322
Traceback	322
THREADLOCAL Common Blocks and ILC with C	323
Example	324

Problem Determination and Debugging 325

Understanding XL Fortran Error Messages	325
Error Severity	325
Compiler Return Code	326
Run-Time Return Code	326
Understanding XL Fortran Messages	326
Limiting the Number of Compile-Time Messages	327
Selecting the Language for Messages	327
Fixing Installation or System Environment Problems	328
Fixing Compile-Time Problems	328
Duplicating Extensions from Other Systems	329
Isolating Problems with Individual Compilation Units	329
Compiling with Thread-safe Commands	329
Running out of Machine Resources	329
Fixing Link-Time Problems	329
Fixing Run-Time Problems	330
Duplicating Extensions from Other Systems	330
Mismatched Sizes or Types for Arguments	330
Working around Problems when Optimizing	330
Input/Output Errors	330
Tracebacks and Core Dumps	331
Debugging a Fortran 90 or Fortran 95 Program	331

Understanding XL Fortran Compiler Listings 333

Header Section	333
Options Section	333
Source Section	334
Error Messages	334
Transformation Report Section	335
Attribute and Cross-Reference Section	336
Object Section	337
File Table Section	337
Compilation Unit Epilogue Section	337
Compilation Epilogue Section	337

Porting Programs to XL Fortran 339

Outline of the Porting Process	339
Portability of Directives	339
NEW	341
Common Industry Extensions That XL Fortran Supports	342
Mixing Data Types in Statements	342
Date and Time Routines	342
Other libc Routines	342
Changing the Default Sizes of Data Types	343
Name Conflicts between Your Procedures and XL Fortran Intrinsic Procedures	343
Reproducing Results from Other Systems	343
Finding Nonstandard Extensions	343

Appendix A. Sample Fortran Programs 345

Example 1 - XL Fortran Source File	345
Execution Results	345
Example 2 - Valid C Routine Source File	346
Example 3 - Valid Fortran SMP Source File	348
Example 4 - Invalid Fortran SMP Source File	348
Programming Examples Using the Pthreads Library Module	349

Appendix B. XL Fortran Technical Information 351

The Compiler Phases	351
External Names in XL Fortran Libraries	351
The XL Fortran Run-Time Environment	351
External Names in the Run-Time Environment	352
Technical Details of the -qfloat=hsflt Option	352
Implementation Details for -qautodbl Promotion and Padding	353
Terminology	353
Examples of Storage Relationships for -qautodbl Suboptions	354

Appendix C. Using the Mathematical Acceleration Subsystem (MASS) 359

Using the Vector Libraries	359
Consistency of MASS Vector Functions	360
Compiling and Linking a Program with MASS	360

Appendix D. XL Fortran Internal Limits 361

Notices 363

Programming Interface Information	365
Trademarks and Service Marks	365

Glossary 367

INDEX 375

Figures

1. Main Fortran Program That Calls C++ (main1.f)	303	7. Storage Relationships with -qautodbl=dbl	355
2. C++ Wrapper Functions for Calling C++ (cfun.C)	303	8. Storage Relationships with -qautobl=dbl4	356
3. C++ Code Called from Fortran (cplus.h)	304	9. Storage Relationships with -qautodbl=dbl8	356
4. Storage Mapping of Parm Area On the Stack in 32-Bit Environment.	319	10. Storage Relationships with -qautodbl=dblpad4.	357
5. Storage Mapping of Parm Area On the Stack in 64-Bit Environment.	320	11. Storage Relationships with -qautodbl=dblpad8.	357
6. Storage Relationships without the -qautodbl Option.	354	12. Storage Relationships with -qautodbl=dblpad	358

What's New for XL Fortran

XL Fortran Version 9.1 provides the following new and changed features:

New or changed compiler options and suboptions:

- The **-qcr** option allows you to control how the compiler interprets the CR (carriage return) character.
- The **-qflttrap=nanq** suboption detects all NaN values handled or generated by floating point instructions, including those not created by invalid operations.
- The **-qport=nullarg** suboption treats an empty argument, which is delimited by a left parenthesis and a comma, two commas, or a comma and a right parenthesis, as a null argument.
- The **-qmodule=mangle81** option provides compatibility with Version 8.1 module naming conventions for non-intrinsic modules.
- The **-qsaveopt** option saves the command-line options used for compiling a source file in the corresponding object file.
- The **-qversion** option provides the version and release for the invoking compiler.
- The **-qhot=vector** suboption converts certain operations that are performed in a loop on successive elements of an array into calls to vector library routines.

The following XL Fortran enhancements adapted from the Fortran 2003 draft standard:

- The **2003std** run-time option provides conformance checking of code for adherence to the draft standard.
- The **ISO_C_BINDING** intrinsic module, **BIND** attribute and statement, module variables, common block, subroutine/function and **-qalign=bindc** compiler suboption provide support for interoperability with C.
- **PUBLIC/PRIVATE** attribute on derived type components.
- The **ASSOCIATE** construct associates an entity with either a variable or the value of an expression.
- Command-line argument intrinsics:
 - **COMMAND_ARGUMENT_COUNT**
 - **GET_COMMAND_ARGUMENT**
 - **GET_ENVIRONMENT_VARIABLE**
- The **FLUSH** statement makes data from an external file available to other processes.
- The **IOMSG=** specifier on the data-transfer operation, file-positioning, **FLUSH**, and file inquiry statements.
- The **ISO_FORTRAN_ENV** intrinsic module provides public entities relating to the Fortran environment.
- The **NEW_LINE** intrinsic returns a new line character.
- The **IMPORT** statement makes named entities from the host scoping unit accessible in the interface body by host association.
- The **PROCEDURE** statement declares a dummy procedure or external procedure.

The following performance-related directives and compiler options/suboptions have been added:

- **-qarch** and **-qtune** compiler suboptions that provide support for POWER5 and PowerPC 970 architectures (ppc64gr, ppc64grsq, pwr5, and ppc970).
- The **-qenablevmx** option instructs the compiler to generate Vector Multimedia eXtension, or VMX, instructions for those PowerPC 970 processors capable of handling them.
- The **-qhot=simd** suboption automatically SIMD vectorizes loops, using Altivec instructions.
- The **-qshowpdf** option, used together with **-qpdf1**, provides additional call and block count profiling information to an executable.
- Optimization utilities **showpdf** and **mergepdf** provide enhanced information about PDF-directed compilation
- The **-qdirectstorage** option informs the compiler that a given compilation unit may reference write-through-enabled or cache-inhibited storage.
- Directives **NOVECTOR**, **NOSIMD**, and the **ALIGNX** built-in subroutine provide fine-grain control of the auto-vectorization and auto-SIMD vectorization features in the compiler.
- The **LOOPID** directive marks a loop with a scope-unique identifier. The identifier can be used by the **BLOCK_LOOP** and other directives to control loop-specific transformations. Information on the loop transformations can be shown in using the **-qreport** compiler of option .
- The **EIEIO** directive helps in with cache and memory management.
- The **PROTECTED STREAM** directives allow for management of protected streams so they are not replaced by any hardware-detected streams.
- The **SWDIV** and **SWDIV_NOCHK** intrinsics provide software floating-point division algorithms.

Other features:

- The **FRE** and **FRSQRTES** PowerPC floating-point intrinsic functions.
- The **POPCNT**, and **POPCNTB** intrinsics provide set bit counts in registers for data objects, and the **POPPAR** intrinsic determines the parity for a data object.
- 32-bit and 64-bit modules are now included in one file.
- Allowing multiple include paths
- Availability of the MASS vector libraries for use with vectorized applications.
- A man page is provided for the compiler invocation commands and for each command-line utility. The man page for compiler invocations replaces the help file, which was provided in previous versions.

Introduction

This document describes IBM® XL Fortran Advanced Edition V9.1 for Linux™ and explains how to compile, link, and run programs that are written in the Fortran language.

How to Use This Document

This document is for anyone who wants to work with the XL Fortran compiler, who is familiar with the Linux operating system, and who has some previous Fortran programming experience.

This document can help you understand what the features of the compiler are, especially the options, and how to use them for effective software development.

This document is not the place to find help on:

Installation,

which is covered in the *IBM XL Fortran Advanced Edition V9.1 for Linux Installation Guide*.

Writing Fortran programs,

which is covered in the *XL Fortran Language Reference*.

The first part of this document is organized according to the steps necessary to compile, link, and run a program, followed by information on particular features of the XL Fortran compiler and the programs it produces.

The second part discusses more general software-development topics.

Depending on your level of experience and what you want to do, you may need to start reading at a particular point or read in a particular sequence. If you want to:

Set up the compiler for yourself or someone else,

read "Where to Find Installation Instructions" on page 11.

Create customized compiler defaults,

read "Customizing the Configuration File" on page 14.

Understand what all the compiler options are for and how they relate to each other,

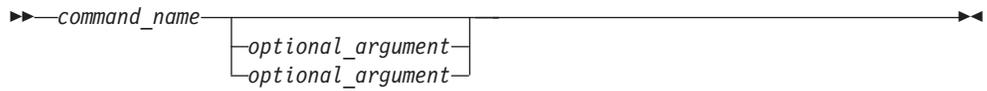
browse through "Summary of the XL Fortran Compiler Options" on page 51.

Look up a particular option by name,

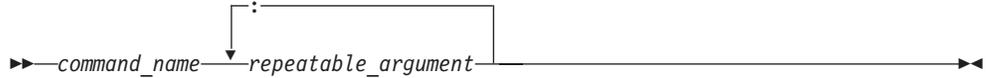
scan alphabetically through "Detailed Descriptions of the XL Fortran Compiler Options" on page 73.

Port a program to XL Fortran,

read "Options for Compatibility" on page 62 to see what options you may need; then read "Porting Programs to XL Fortran" on page 339 for other porting information.

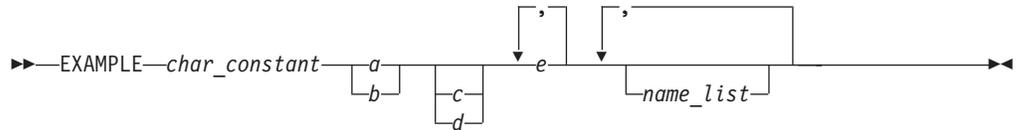


- An arrow returning to the left above the main line (a "repeat arrow") indicates an item that can be repeated and the separator character if it is other than a blank:



A repeat arrow above a stack indicates that you can make more than one choice from the stacked items.

Example of a Syntax Diagram



Interpret the diagram as follows:

- 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*.

Syntax Statements

Syntax statements are read from left to right:

- Individual required arguments are shown with no special notation.
- When you must make a choice between a set of alternatives, they are enclosed by { and } symbols.
- Optional arguments are enclosed by [and] symbols.
- When you can select from a group of choices, they are separated by | characters.
- Arguments that you can repeat are followed by ellipses (...).

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.

Notes on the Examples in This Document

- The examples in this document are coded in a simple style that does not try to conserve storage, check for errors, achieve fast performance, or demonstrate all possible ways to do something.
- The examples in this document use the **xlf90**, **xlf90_r**, **xlf95**, **xlf95_r**, **xlf**, **xlf_r**, **f77**, **fort77**, **f90**, and **f95** compiler invocation commands interchangeably. For more substantial source files, one of these commands may be more suitable than the others, as explained in “Compiling XL Fortran Programs” on page 21.
- Some sample programs from this document and some other programs that illustrate ideas presented in this document are in the directory `/opt/ibmcmp/xlf/9.1/samples`.

Notes on the Path Names in This Document

The path names shown in this document assume the default installation path for the XL Fortran compiler. By default, XL Fortran will be installed in the following directory on the selected disk:

```
/opt/ibmcmp/xlf/9.1
```

You can select a different destination (*relocation-path*) for the compiler. If you choose a different path, the compiler will be installed in the following directory:

```
relocation-path/opt/ibmcmp/xlf/9.1
```

Notes on the Terminology in This Document

Some of the terminology in this document is shortened, as follows:

- The term *free source form format* will often appear as *free source form*.
- The term *fixed source form format* will often appear as *fixed source form*.
- The term *XL Fortran* will often appear as *XLF*.

Typographical Conventions

This document uses the following methods to differentiate text:

- Fortran keywords, commands, statements, directives, intrinsic procedures, compiler options, and filenames are shown in **bold**. For example, **COMMON**, **END**, and **OPEN**.
- References to other sources of information appear in *italics*.
- Variable names and user-specified names appear in lowercase italics. For example, *array_element_name*.

Related Documentation

You can refer to the following publications for additional information:

XL Fortran and Operating System Publications

- *IBM XL Fortran Advanced Edition V9.1 for Linux Language Reference* describes the XL Fortran programming language.
- The *RPM home page* at URL <http://www.rpm.org/> covers all aspects of the standard Linux installation procedure using the RPM Package Manager (RPM). XL Fortran supplies brief installation instructions in the *IBM XL Fortran Advanced*

Edition V9.1 for Linux Installation Guide that explain how the general-purpose installation procedures apply to this licensed program.

- For general information and documentation on Linux, visit *The Linux Documentation Project* at URL <http://www.tldp.org/>. Consult your Linux-specific distribution's man page help and documentation for information on using your operating system and its features.
- Visit the *Linux at IBM* home page at URL <http://www.ibm.com/linux/> for information on IBM-related offerings for Linux.
- *System V Application Binary Interface: PowerPC Processor Supplement* is a supplement to the generic System V ABI and contains information specific to System V implementations built on the PowerPC Architecture™ operating in 32-bit mode.
- *64-bit PowerPC ELF Application Binary Interface Supplement* is a supplement to the generic System V ABI and contains information specific to System V implementations built on the PowerPC Architecture operating in 64-bit mode.

Standards Documents

You may want to refer to these standards for precise definitions of some of the features referred to in this document:

- *American National Standard Programming Language FORTRAN*, ANSI X3.9-1978.
- *American National Standard Programming Language Fortran 90*, ANSI X3.198-1992. (Referred to in this document by its informal name, Fortran 90.)
- *Federal (USA) Information Processing Standards Publication Fortran*, FIPS PUB 69-1.
- *ANSI/IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Std 754-1985.
- *Information technology - Programming languages - Fortran*, ISO/IEC 1539-1:1991(E).
- *Information technology - Programming languages - Fortran - Part 1: Base language*, ISO/IEC 1539-1:1997. (Referred to in this document by its informal name, Fortran 95.)
- *Information technology - Programming Languages - Fortran - Floating-Point Exception Handling*, ISO/IEC JTC1/SC22/WG5 N1379.
- *Information technology - Programming Languages - Fortran - Enhanced Data Type Facilities*, ISO/IEC JTC1/SC22/WG5 N1378.
- *Military Standard Fortran DOD Supplement to ANSI X3.9-1978*, MIL-STD-1753 (United States of America, Department of Defence standard). Note that XL Fortran supports only those extensions that have been subsequently incorporated into the Fortran 90 and Fortran 95 standards.
- *OpenMP Fortran Application Program Interface, Version 2.0, (Nov 2000)*. (Referred to in this document by its informal name, OpenMP Fortran API.)

Overview of XL Fortran Features

This section discusses the features of the XL Fortran compiler, language, and development environment at a high level. It is intended for people who are evaluating XL Fortran and for new users who want to find out more about the product.

Hardware and Operating-System Support

The XL Fortran Advanced Edition Version 9.1 compiler is supported on several Linux distributions. See the *IBM XL Fortran Advanced Edition V9.1 for Linux Installation Guide* and README file for a list of supported distributions and requirements.

The compiler, its generated object programs, and run-time library will run on all POWER3™, POWER4™, POWER5, PowerPC 970, and PowerPC® systems with the required software, disk space, and virtual storage.

The POWER3, POWER4, or POWER5 processor is a type of PowerPC. In this document, any statement or reference to the PowerPC also applies to the POWER3, POWER4, or POWER5 processor.

To take maximum advantage of different hardware configurations, the compiler provides a number of options for performance tuning based on the configuration of the machine used for executing an application.

Language Support

The XL Fortran language consists of the following:

- The full American National Standard Fortran 90 language (referred to as Fortran 90 or F90), defined in the documents *American National Standard Programming Language Fortran 90*, ANSI X3.198-1992 and *Information technology - Programming languages - Fortran*, ISO/IEC 1539-1:1991(E). This language has a superset of the features found in the FORTRAN 77 standard. It adds many more features that are intended to shift more of the tasks of error checking, array processing, memory allocation, and so on from the programmer to the compiler.
- The full ISO Fortran 95 language standard (referred to as Fortran 95 or F95), defined in the document *Information technology - Programming languages - Fortran - Part 1: Base language*, ISO/IEC 1539-1:1997.
- Extensions to the Fortran 95 standard:
 - Industry extensions that are found in Fortran products from various compiler vendors
 - Extensions specified in SAA Fortran

In the *XL Fortran Language Reference*, extensions to the Fortran 95 language are marked as described in the *Typographical Conventions* topic.

Migration Support

The XL Fortran compiler helps you to port or to migrate source code among Fortran compilers by providing full Fortran 90 and Fortran 95 language support and selected language extensions (intrinsic functions, data types, and so on) from many different compiler vendors. Throughout this document, we will refer to these extensions as “industry extensions”.

To protect your investment in FORTRAN 77 source code, you can easily invoke the compiler with a set of defaults that provide backward compatibility with earlier versions of XL Fortran. The `xlf`, `xlf_r`, `f77`, and `fort77` commands provide maximum compatibility with existing FORTRAN 77 programs. The default options provided with the `xlf90` and `xlf90_r` commands give access to the full range of Fortran 90 language features. The default options provided with the `xlf95` and `xlf95_r` commands give access to the full range of Fortran 95 language features.

Source-Code Conformance Checking

To help you find anything in your programs that might cause problems when you port to or from different FORTRAN 77, Fortran 90, or Fortran 95 compilers, the XL Fortran compiler provides options that warn you about features that do not conform to certain Fortran definitions.

If you specify the appropriate compiler options, the XL Fortran compiler checks source statements for conformance to the following Fortran language definitions:

- Full American National Standard FORTRAN 77 (`-qlanglvl=77std` option), full American National Standard Fortran 90 (`-qlanglvl=90std` option), and full Fortran 95 standard (`-qlanglvl=95std` option)
- Fortran 90, less any obsolescent features (`-qlanglvl=90pure` option)
- Fortran 95, less any obsolescent features (`-qlanglvl=95pure` option)
- IBM SAA[®] FORTRAN (`-qsaa` option)

You can also use the `langlvl` environment variable for conformance checking.

Highly Configurable Compiler

You can invoke the compiler by using the `xlf`, `xlf_r`, `xlf90`, `xlf90_r`, `xlf95`, `xlf95_r`, `f77`, or `fort77` command. The `xlf`, `xlf_r`, and `f77` commands maintain maximum compatibility with the behavior and I/O formats of XL Fortran Version 2. The `xlf90` and `xlf90_r` commands provide more Fortran 90 conformance and some implementation choices for efficiency and usability. The `xlf95` and `xlf95_r` commands provide more Fortran 95 conformance and some implementation choices for efficiency and usability. The `fort77` command provides maximum compatibility with the XPG4 behavior.

The main difference between the set of `xlf_r`, `xlf90_r`, and `xlf95_r` commands and the set of `xlf`, `xlf90`, `xlf95`, `f77`, and `fort77` commands is that the first set links and binds the object files to the thread-safe components (libraries, and so on). You can have this behavior with the second set of commands by using the `-F` compiler option to specify the configuration file stanza to use. For example:

```
xlf -F/etc/opt/ibmcmp/xlf/9.1/xlf.cfg:xlf_r
```

You can control the actions of the compiler through a set of options. The different categories of options help you to debug, to optimize and tune program

performance, to select extensions for compatibility with programs from other platforms, and to do other common tasks that would otherwise require changing the source code.

To simplify the task of managing many different sets of compiler options, you can customize the single file `/etc/opt/ibmcomp/xf/9.1/xf.cfg` instead of creating many separate aliases or shell scripts.

For information on:

- The configuration file, see “Customizing the Configuration File” on page 14
- The invocation commands, see “Compiling XL Fortran Programs” on page 21
- The compiler options, see “Summary of the XL Fortran Compiler Options” on page 51 and “Detailed Descriptions of the XL Fortran Compiler Options” on page 73
- Compiler return codes, see “Understanding XL Fortran Messages” on page 326

Diagnostic Listings

The compiler output listing has optional sections that you can include or omit. For information about the applicable compiler options and the listing itself, refer to “Options That Control Listings and Messages” on page 61 and “Understanding XL Fortran Compiler Listings” on page 333.

The `-S` option gives you a true assembler source file.

Symbolic Debugger Support

You can use `gdb` and other symbolic debuggers for your programs.

Program Optimization

The XL Fortran compiler helps you control the optimization of your programs:

- You can select different levels of compiler optimizations.
- You can turn on separate optimizations for loops, floating point, and other categories.
- You can optimize a program for a particular class of machines or for a very specific machine configuration, depending on where the program will run.

“Optimizing XL Fortran Programs” on page 265 is a road map to these features.

Documentation and Online Help

XL Fortran provides product documentation in the following formats:

- Readme files
- Installable man pages
- An HTML-based information system
- Portable Document Format (PDF) documents

These items are located, or accessed as follows:

Readme files The readme files are located in `/opt/ibmcomp/xf/9.1/doc/$LANG/` and in the root directory of the installation CD.

Man pages Man pages are provided for the compiler invocations and all command-line utilities provided with the product.

HTML-based information

An information system composed of HTML files is provided in the

`/opt/ibmcmp/xlf/9.1/doc/$LANG/html` directory. To access the HTML information, open the `index.htm` file in a frames-capable browser.

PDF documents

The PDF files are located in the `/opt/ibmcmp/xlf/9.1/doc/$LANG/pdf` directory. They are viewable and printable from a pdf viewer like Xpdf.

LANG represents the language and location code. For example, `en_US` is the language and location code for US English.

To view the documentation, you must have the X Window system working on your system. In addition, you need to have the appropriate viewers, such as a frames-capable HTML browser like Mozilla (version 1.0, or later) and a PDF viewer, such as Xpdf.

Cross references between important User's Guide and Language Reference topics are linked. While viewing a given document, you can access a linked topic in the other document by clicking the link. You do not need to close the document you are viewing to access the linked information in the other document.

For the latest information about XL Fortran Advanced Edition, visit the product web site at:

- <http://www.ibm.com/software/awdtools/fortran/xlfortran/support>.

Setting Up and Customizing XL Fortran

This section explains how to customize XL Fortran settings for yourself or all users and how to set up a user account to use XL Fortran. The full installation procedure is beyond the scope of this section, which refers you to the documents that cover the procedure in detail.

This section can also help you to diagnose problems that relate to installing or configuring the compiler.

Some of the instructions require you to be a superuser, and so they are only applicable if you are a system administrator.

Where to Find Installation Instructions

To install the compiler, refer to these documents (preferably in this order):

1. Read the file called `/opt/ibmcmp/xlf/9.1/doc/en_US/README`, and follow any directions it gives. It contains information that you should know and possibly distribute to other people who use XL Fortran.
2. Read any *Installation Guide* document that comes with the compiler to see if there are any important notices you should be aware of or any updates you might need to apply to your system before doing the installation.
3. You should be familiar with the RPM Package Manager (RPM) for installing this product. For information on using RPM, visit the RPM Web page at URL <http://www.rpm.org/>, or type `rpm --help` at the command line.

If you are already experienced with Linux software installation, you can use the `rpm` command to install all the images from the distribution medium.

Using the Compiler on a Network File System

If you want to use the XL Fortran compiler on a Network File System server for a networked cluster of machines, use the Network Install Manager.

The following directories contain XL Fortran components:

- `/opt/ibmcmp/xlf/9.1/bin` contains the compiler invocation commands.
- `/opt/ibmcmp/xlf/9.1/exe` contains executables and files that the compiler needs.
- `/opt/ibmcmp/xlf/9.1/lib` and `/opt/ibmcmp/xlf/9.1/lib64` contain the non-redistributable libraries.
- `/opt/ibmcmp/lib/` and `/opt/ibmcmp/lib64/` contain the redistributable libraries.
- `/opt/ibmcmp/xlf/9.1/include` contains the include files and supplied `.mod` files.
- `/opt/ibmcmp/msg` contains the message catalogues for the redistributable run-time libraries.
- `/opt/ibmcmp/xlf/9.1/doc/en_US/pdf` contains the PDF format of the English XL Fortran publications.
`/opt/ibmcmp/xlf/9.1/doc/ja_JP/pdf` contains the PDF format of the Japanese XL Fortran publications.
- `/opt/ibmcmp/xlf/9.1/doc/en_US/html` contains the HTML format of the English XL Fortran publications.
`/opt/ibmcmp/xlf/9.1/doc/ja_JP/html` contains the HTML format of the Japanese XL Fortran publications.

You must also copy the `/etc/opt/ibmcomp/xlf/9.1/xlf.cfg` file from the server to the client. The `/etc/opt/ibmcomp/xlf/9.1` directory contains the configuration files specific to a machine, and it should not be mounted from the server.

Correct Settings for Environment Variables

You can set and export a number of environment variables for use with the operating system. The following sections deal with the environment variables that have special significance to the XL Fortran compiler, application programs, or both.

Environment Variable Basics

You can set the environment variables from shell command lines or from within shell scripts. (For more information about setting environment variables, see the man page help for the shell you are using.) If you are not sure which shell is in use, a quick way to find out is to issue `echo $SHELL` to show the name of the current shell.

To display the contents of an environment variable, enter the command `echo $var_name`.

Note: For the remainder of this document, most examples of shell commands use **Bash** notation instead of repeating the syntax for all shells.

Environment Variables for National Language Support

Diagnostic messages and the listings from the compiler appear in the default language that was specified at installation of the operating system. If you want the messages and listings to appear in another language, you can set and export the following environment variables before executing the compiler:

- | | |
|----------------|--|
| LANG | Specifies the <i>locale</i> . A locale is divided into categories. Each category contains a specific aspect of the locale data. Setting LANG may change the national language for all the categories. |
| NLSPATH | Refers to a list of directory names where the message catalogs may be found. |

For example, to specify the Japanese locale, set the **LANG** environment variable to **ja_JP**.

Substitute any valid national language code for **ja_JP**, provided the associated message catalogs are installed.

These environment variables are initialized when the operating system is installed and may be different from the ones that you want to use with the compiler.

Each category has an environment variable associated with it. If you want to change the national language for a specific category but not for other categories, you can set and export the corresponding environment variable.

For example:

LC_MESSAGES

Specifies the national language for the messages that are issued. It affects messages from the compiler and XLF-compiled programs, which may be displayed on the screen or stored in a listing, module, or other compiler output file.

LC_TIME

Specifies the national language for the time format category. It primarily affects the compiler listings.

LC_CTYPE

Defines character classification, case conversion, and other character attributes. For XL Fortran, it primarily affects the processing of multibyte characters.

LC_NUMERIC

Specifies the format to use for input and output of numeric values. Setting this variable in the shell *does not* affect either the compiler or XLF-compiled programs. The first I/O statement in a program sets the **LC_NUMERIC** category to **POSIX**. Therefore, programs that require a different setting must reset it after this point and should restore the setting to **POSIX** for all I/O statements.

Notes:

1. Specifying the **LC_ALL** environment variable overrides the value of the **LANG** and other **LC_** environment variables.
2. If the XL Fortran compiler or application programs cannot access the message catalogs or retrieve a specific message, the message appears in U.S. English.
3. The backslash character, `\`, has the same hexadecimal code, `X'5C'`, as the Yen symbol and can appear as the Yen symbol on the display if the locale is Japanese.

Related Information: “Selecting the Language for Run-Time Messages” on page 34.

See the Linux-specific documentation and man page help for more information about National Language Support environment variables and locale concepts.

Setting Library Search Paths

If your executable program is linked with shared libraries, you need to set the run-time library search paths. There are two ways to set run-time library search paths. You can use:

- The **-R** (or **-rpath**) compile/link option, or
- The **LD_LIBRARY_PATH** and **LD_RUN_PATH** environment variables

Specifying search paths with the compile/link **-R** (or **-rpath**) option has the effect of writing the specified run-time library search paths into the executable program. If you use the **-L** option, library search paths are searched at link time, but are not written into the executable as run-time library search paths. For example:

```
# Compile and link
xlf95 -L/usr/lib/mydir1 -R/usr/lib/mydir1 -L/usr/lib/mydir2 -R/usr/lib/mydir2
-lmylib1 -lmylib2 test.f
```

```
# -L directories are searched at link time.
# -R directories are searched at run time.
```

You can also use the **LD_LIBRARY_PATH** and **LD_RUN_PATH** environment variables to specify library search paths. Use **LD_RUN_PATH** to specify the directories that are to be searched for libraries at run time. Use **LD_LIBRARY_PATH** to specify the directories that are to be searched for libraries at both link and run time.

For more information on linker options and environment variables, see the man pages for the `ld` command.

PDFDIR: Specifying the Directory for PDF Profile Information

When you compile a Fortran 90 program with the `-qpdf` compiler option, you can specify the directory where profiling information is stored by setting the `PDFDIR` environment variable to the name of the directory. The compiler creates the files to hold the profile information. XL Fortran updates the files when you run an application that is compiled with the `-qpdf1` option.

Because problems can occur if the profiling information is stored in the wrong place or is updated by more than one application, you should follow these guidelines:

- Always set the `PDFDIR` variable when using the `-qpdf` option.
- Store the profiling information for each application in a different directory, or use the `-qipa=pdfname=[filename]` option to explicitly name the temporary profiling files according to the template provided.
- Leave the value of the `PDFDIR` variable the same until you have completed the PDF process (compiling, running, and compiling again) for the application.

TMPDIR: Specifying a Directory for Temporary Files

The XL Fortran compiler creates a number of temporary files for use during compilation. An XL Fortran application program creates a temporary file at run time for a file opened with `STATUS='SCRATCH'`. By default, these files are placed in the directory `/tmp`.

If you want to change the directory where these files are placed, perhaps because `/tmp` is not large enough to hold all the temporary files, set and export the `TMPDIR` environment variable before running the compiler or the application program.

If you explicitly name a scratch file by using the `XLFSCRATCH_unit` method described below, the `TMPDIR` environment variable has no effect on that file.

XLFSCRATCH_unit: Specifying Names for Scratch Files

To give a specific name to a scratch file, you can set the run-time option `scratch_vars=yes`; then set one or more environment variables with names of the form `XLFSCRATCH_unit` to file names to use when those units are opened as scratch files. See “Naming Scratch Files” on page 291 for examples.

XLFUNIT_unit: Specifying Names for Implicitly Connected Files

To give a specific name to an implicitly connected file or a file opened with no `FILE=` specifier, you can set the run-time option `unit_vars=yes`; then set one or more environment variables with names of the form `XLFUNIT_unit` to file names. See “Naming Files That Are Connected with No Explicit Name” on page 290 for examples.

Customizing the Configuration File

The configuration file specifies information that the compiler uses when you invoke it. XL Fortran provides the default configuration file `/etc/opt/ibmcmp/xf/9.1/xf.cfg` at installation time.

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

Otherwise, especially if you want many users to be able to choose among several sets of compiler options, you may want to add new named stanzas to the configuration file and to create new commands that are links to existing commands. For example, you could specify something similar to the following to create a link to the `xlf95` command:

```
ln -s /opt/ibmcomp/xlf/9.1/bin/xlf95 /home/lisa/bin/my_xlf95
```

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

Notes:

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

Attributes

The configuration file contains the following attributes:

use	The named and local stanzas provide the values for attributes. For single-valued attributes, values in the use attribute apply if there is no value in the local, or default, stanza. For comma-separated lists, the values from the use attribute are added to the values from the local stanza. You can only use a single level of the use attribute. Do not specify a use attribute that names a stanza with another use attribute.
crt	When invoked in 32-bit mode, the default (which is the path name of the object file that contains the startup code), passed as the first parameter to the linkage editor.
crt_64	When invoked in 64-bit mode, using -q64 for example, the path name of the object file that contains the startup code, passed as the first parameter to the linkage editor.
mcrt	Same as for crt , but the object file contains profiling code for the -p option.
mcrt_64	Same as for crt_64 , but the object file contains profiling code for the -p option.
gcrt	Same as crt , but the object file contains profiling code for the -pg option.
gcrt_64	Same as crt_64 , but the object file contains profiling code for the -pg option.
gcc_libs	When invoked in 32-bit mode, the linker options to specify the path to the GCC libraries and to link the GCC library.
gcc_libs_64	When invoked in 64-bit mode, the linker options to specify the path to the GCC libraries and to link the GCC library.

gcc_path	Specifies the path to the 32-bit tool chain.
gcc_path_64	Specifies the path to the 64-bit tool chain.
cpp	The absolute path name of the C preprocessor, which is automatically called for files ending with a specific suffix (usually .F).
xlf	The absolute path name of the main compiler executable file. The compiler commands are driver programs that execute this file.
code	The absolute path name of the optimizing code generator.
xlfopt	Lists names of options that are assumed to be compiler options, for cases where, for example, a compiler option and a linker option use the same letter. The list is a concatenated set of single-letter flags. Any flag that takes an argument is followed by a colon, and the whole list is enclosed by double quotation marks.
as	The absolute path name of the assembler.
asopt	Lists names of options that are assumed to be assembler options for cases where, for example, a compiler option and an assembler option use the same letter. The list is a concatenated set of single-letter flags. Any flag that takes an argument is followed by a colon, and the whole list is enclosed by double quotation marks. You may find it more convenient to set up this attribute than to pass options to the assembler through the -W compiler option.
ld	The absolute path name of the linker.
ldopt	Lists names of options that are assumed to be linker options for cases where, for example, a compiler option and a linker option use the same letter. The list is a concatenated set of single-letter flags. Any flag that takes an argument is followed by a colon, and the whole list is enclosed by double quotation marks. You may find it more convenient to set up this attribute than to pass options to the linker through the -W compiler option. However, most unrecognized options are passed to the linker anyway.
options	A string of options that are separated by commas. The compiler processes these options as if you entered them on the command line before any other option. This attribute lets you shorten the command line by including commonly used options in one central place.
cppoptions	A string of options that are separated by commas, to be processed by cpp (the C preprocessor) as if you entered them on the command line before any other option. This attribute is needed because some cpp options are usually required to produce output that can be compiled by XL Fortran. The default option is -C , which preserves any C-style comments in the output.
fsuffix	The allowed suffix for Fortran source files. The default is f . The compiler requires that all source files in a single compilation have the same suffix. Therefore, to compile files with other suffixes, such as f95 , you must change this attribute in the configuration file or use the -qsuffix compiler option. For more information on -qsuffix , see “-qsuffix Option” on page 209.

cppsuffix	The suffix that indicates a file must be preprocessed by the C preprocessor (cpp) before being compiled by XL Fortran. The default is F .
osuffix	The suffix used to recognize object files that are specified as input files. The default is o .
ssuffix	The suffix used to recognize assembler files that are specified as input files. The default is s .
libraries	-l options, which are separated by commas, that specify the libraries used to link all programs.
smplibraries	Specifies the libraries that are used to link programs that you compiled with the -qsmp compiler option.
hot	Absolute path name of the program that does array language transformations.
ipa	Absolute path name of the program that performs interprocedural optimizations, loop optimizations, and program parallelization.
bolt	Absolute path name of the binder.
defaultmsg	Absolute path name of the default message files.
include	Indicates the search path that is used for compilation include files and module files.
include_32	Indicates the search path that is used for 32-bit compilation include files.
include_64	Indicates the search path that is used for 64-bit compilation include files.

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

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

What a Configuration File Looks Like

The following is an example of a configuration file:

```
xlf95:  use          = DEFLT
        libraries    = -xlf90,-lxlopt,-xlomp_ser,-lxl,-lxfmath
        libraries_64 = -xlf90,-lxlopt,-xlomp_ser,-lxl,-lxfmath
        gcc_libs     = -ldl,-lrt,-lpthread,-lm,-lc,-lgcc
        gcc_libs_64  = -ldl,-lrt,-lpthread,-lm,-lc,-lgcc
        options      = -qfree=f90

* Alias for standard Fortran compiler
f95:    use          = DEFLT
        libraries    = -xlf90,-lxlopt,-xlomp_ser,-lxl,-lxfmath
        libraries_64 = -xlf90,-lxlopt,-xlomp_ser,-lxl,-lxfmath
        gcc_libs     = -ldl,-lrt,-lpthread,-lm,-lc,-lgcc
        gcc_libs_64  = -ldl,-lrt,-lpthread,-lm,-lc,-lgcc
        options      = -qfree=f90
        fsuffix      = f95

* Fortran 90 compiler
xlf90:  use          = DEFLT
        libraries    = -xlf90,-lxlopt,-xlomp_ser,-lxl,-lxfmath
        libraries_64 = -xlf90,-lxlopt,-xlomp_ser,-lxl,-lxfmath
        gcc_libs     = -ldl,-lrt,-lpthread,-lm,-lc,-lgcc
        gcc_libs_64  = -ldl,-lrt,-lpthread,-lm,-lc,-lgcc
        options      = -qxlf90=noautoalloc:nosignedzero,-qfree=f90
```

```

* Alias for Fortran 90 compiler
f90:    use      = DEFLT
        libraries = -lxlf90,-lxlopt,-lxlomp_ser,-lxl,-lxlfmath
        libraries_64 = -lxlf90,-lxlopt,-lxlomp_ser,-lxl,-lxlfmath
        gcc_libs    = -ldl,-lrt,-lpthread,-lm,-lc,-lgcc
        gcc_libs_64 = -ldl,-lrt,-lpthread,-lm,-lc,-lgcc
        options     = -qxlf90=noautodealloc:nosignedzero,-qfree=f90
        fsuffix     = f90

* Original Fortran compiler
xlf:    use      = DEFLT
        libraries = -lxlf90,-lxlopt,-lxlomp_ser,-lxl,-lxlfmath
        libraries_64 = -lxlf90,-lxlopt,-lxlomp_ser,-lxl,-lxlfmath
        gcc_libs    = -ldl,-lrt,-lpthread,-lm,-lc,-lgcc
        gcc_libs_64 = -ldl,-lrt,-lpthread,-lm,-lc,-lgcc
        options     = -qnozerosize,-qsave,-qalias=intptr,
                    -qposition=appendold,-qxlf90=noautodealloc:nosignedzero,
                    -qxlf77=intarg:intxor:persistent:noleadzero:gedit77:
                    noblankpad:oldboz:softeof

* Alias for original Fortran compiler
f77:    use      = DEFLT
        libraries = -lxlf90,-lxlopt,-lxlomp_ser,-lxl,-lxlfmath
        libraries_64 = -lxlf90,-lxlopt,-lxlomp_ser,-lxl,-lxlfmath
        gcc_libs    = -ldl,-lrt,-lpthread,-lm,-lc,-lgcc
        gcc_libs_64 = -ldl,-lrt,-lpthread,-lm,-lc,-lgcc
        options     = -qnozerosize,-qsave,-qalias=intptr,
                    -qposition=appendold,
                    -qxlf90=noautodealloc:nosignedzero,-qxlf77=intarg:
                    intxor:persistent:noleadzero:gedit77:noblankpad:
                    oldboz:softeof

* Alias for original Fortran compiler, used for XPG4 compliance
fort77: use      = DEFLT
        libraries = -lxlf90,-lxlopt,-lxlomp_ser,-lxl,-lxlfmath
        libraries_64 = -lxlf90,-lxlopt,-lxlomp_ser,-lxl,-lxlfmath
        gcc_libs    = -ldl,-lrt,-lpthread,-lm,-lc,-lgcc
        gcc_libs_64 = -ldl,-lrt,-lpthread,-lm,-lc,-lgcc
        options     = -qnozerosize,-qsave,-qalias=intptr,-qposition=appendold,
                    -qxlf90=noautodealloc:nosignedzero,-qxlf77=intarg:intxor:
                    persistent:noleadzero:gedit77:noblankpad:oldboz:softeof

* xlf with links to thread-safe components
xlf_r:  use      = DEFLT
        libraries = -lxlf90_r,-lxlopt,-lxlomp_ser,-lxl,-lxlfmath
        libraries_64 = -lxlf90_r,-lxlopt,-lxlomp_ser,-lxl,-lxlfmath
        smplibraries = -lxlf90_r,-lxlopt,-lxlsmp,-lxl,-lxlfmath
        smplibraries_64 = -lxlf90_r,-lxlopt,-lxlsmp,-lxl,-lxlfmath
        gcc_libs    = -ldl,-lrt,-lpthread,-lm,-lc,-lgcc
        gcc_libs_64 = -ldl,-lrt,-lpthread,-lm,-lc,-lgcc
        options     = -qthreaded,-qnozerosize,-qsave,-qalias=intptr,
                    -qposition=appendold,-qxlf90=noautodealloc:nosignedzero,
                    -qxlf77=intarg:intxor:persistent:noleadzero:gedit77:
                    noblankpad:oldboz:softeof

* xlf90 with links to thread-safe components
xlf90_r: use      = DEFLT
        libraries = -lxlf90_r,-lxlopt,-lxlomp_ser,-lxl,-lxlfmath
        libraries_64 = -lxlf90_r,-lxlopt,-lxlomp_ser,-lxl,-lxlfmath
        smplibraries = -lxlf90_r,-lxlopt,-lxlsmp,-lxl,-lxlfmath
        smplibraries_64 = -lxlf90_r,-lxlopt,-lxlsmp,-lxl,-lxlfmath
        gcc_libs    = -ldl,-lrt,-lpthread,-lm,-lc,-lgcc
        gcc_libs_64 = -ldl,-lrt,-lpthread,-lm,-lc,-lgcc
        options     = -qxlf90=noautodealloc:nosignedzero,-qfree=f90,
                    -qthreaded

```

```

* xlf95 with links to thread-safe components
xlf95_r: use          = DEFLT
          libraries   = -lxlf90_r,-lxlopt,-lxlomp_ser,-lxl,-lxlfmath
          libraries_64 = -lxlf90_r,-lxlopt,-lxlomp_ser,-lxl,-lxlfmath
          smplibraries = -lxlf90_r,-lxlopt,-lxlsmp,-lxl,-lxlfmath
          smplibraries_64 = -lxlf90_r,-lxlopt,-lxlsmp,-lxl,-lxlfmath
          gcc_libs     = -ldl,-lrt,-lpthread,-lm,-lc,-lgcc
          gcc_libs_64  = -ldl,-lrt,-lpthread,-lm,-lc,-lgcc
          options      = -qfree=f90,-qthreaded

* Common definitions
DEFLT: xlf          = /opt/ibmcmp/xlf/9.1/exe/xlfentry
       crt          = /usr/lib/crt1.o
       crtp         = /usr/lib/crti.o
       crte         = /usr/lib/crtn.o
       crtbegin     = /usr/lib/gcc-lib/powerpc-suse-linux/3.3.3/crtbegin.o
       crtend       = /usr/lib/gcc-lib/powerpc-suse-linux/3.3.3/crtend.o
       crtsavres    = /usr/lib/gcc-lib/powerpc-suse-linux/3.3.3/crtsavres.o
       mcrt         = /usr/lib/gcrt1.o
       gcrt         = /usr/lib/gcrt1.o
       crt_64       = /usr/lib64/crt1.o
       crtp_64      = /usr/lib64/crti.o
       crte_64      = /usr/lib64/crtn.o
       crtbegin_64  = /usr/lib/gcc-lib/powerpc-suse-linux/3.3.3/64/crtbegin.o
       crtend_64    = /usr/lib/gcc-lib/powerpc-suse-linux/3.3.3/64/crtend.o
       crtsavres_64 = /usr/lib/gcc-lib/powerpc-suse-linux/3.3.3/64/crtsavres.o
       mcrt_64      = /usr/lib64/gcrt1.o
       gcrt_64      = /usr/lib64/gcrt1.o
       include_32   = -I/opt/ibmcmp/xlf/9.1/include
       include_64   = -I/opt/ibmcmp/xlf/9.1/include64
       dis          = /opt/ibmcmp/xlf/9.1/exe/dis
       code         = /opt/ibmcmp/xlf/9.1/exe/xlfcode
       hot          = /opt/ibmcmp/xlf/9.1/exe/xlfhot
       ipa          = /opt/ibmcmp/xlf/9.1/exe/ipa
       bolt         = /opt/ibmcmp/xlf/9.1/exe/bolt
       defaultmsg   = /opt/ibmcmp/xlf/9.1/msg/en_US
       as           = /usr/bin/as
       as_64        = /usr/bin/as
       ld           = /usr/bin/ld
       ld_64        = /usr/bin/ld
       cppoptions   = -C
       cpp          = /opt/ibmcmp/xlf/9.1/exe/cpp
       dynlib       = -dynamic-linker,/lib/ld.so.1
       dynlib_64    = -dynamic-linker,/lib64/ld64.so.1
       libdirs      = -L/opt/ibmcmp/xlsmp/1.3/lib,-L/opt/ibmcmp/xlf/9.1/lib,
                    -R/opt/ibmcmp/xlsmp/1.3/../../lib,
                    -R/opt/ibmcmp/xlf/9.1/../../lib
       libdirs_64   = -L/opt/ibmcmp/xlsmp/1.3/lib64,-L/opt/ibmcmp/xlf/9.1/lib64,
                    -R/opt/ibmcmp/xlsmp/1.3/../../lib64,
                    -R/opt/ibmcmp/xlf/9.1/../../lib64
       gcc_path     = /usr
       gcc_path_64  = /usr
       gcc_libdirs  = -L/usr/lib/gcc-lib/powerpc-suse-linux/3.3.3,
                    -L/usr/powerpc-suse-linux/lib,
                    -L/usr/lib,-L/lib,
       gcc_libdirs_64 = -L/usr/lib/gcc-lib/powerpc-suse-linux/3.3.3/64,
                    -L/usr/lib/gcc-lib/powerpc-suse-linux/3.3.3,
                    -L/usr/powerpc-suse-linux/lib,
                    -L/usr/lib,-L/lib64,
                    -L/usr/lib64
       xlcmp_path   = /opt/ibmcmp/xlf/9.1
       bigdata      = -T/opt/ibmcmp/xlf/9.1/exe/elf32ppclinux.x
       bigdata_shr  = -T/opt/ibmcmp/xlf/9.1/exe/elf32ppclinux_shr.x
       options      = -qnoenablevmx

```

```
modes_configure = 32_64
crt2 = NULL
crt2_64 = NULL
xlf_path = /opt/ibmcmp/xlf/9.1
```

XL Fortran provides the library **libxlf90_r.so** in addition to **libxlf90_t.so**. The library **libxlf90_r.so** is a superset of **libxlf90_t.so**, which is a partial thread-support run-time library. The file **xlf.cfg** has been set up to link to **libxlf90_r.so** automatically when you use the **xlf90_r**, **xlf95_r**, and **xlf_r** commands.

Related Information: You can use the “-F Option” on page 81 to select a different configuration file, a specific stanza in the configuration file, or both.

Determining Which Level of XL Fortran Is Installed

Sometimes, you may not be sure which level of XL Fortran is installed on a particular machine. You would need to know this information before contacting software support.

To check whether the latest level of the product has been installed through the system installation procedure, issue the command:

```
rpm -qa | grep xlf.cmp-9.1 | xargs rpm -qi
```

The result includes the version, release, modification, and fix level of the compiler image installed on the system.

You can also use the **-qversion** compiler option to see the version and release for the compiler.

Running Two Levels of XL Fortran

It is possible for two different levels of the XL Fortran compiler to coexist on one system. This allows you to invoke one level by default and to invoke the other one whenever you explicitly choose to.

To do this, consult the *IBM XL Fortran Advanced Edition V9.1 for Linux Installation Guide* for details.

Editing, Compiling, Linking, and Running XL Fortran Programs

Most Fortran program development consists of a repeating cycle of editing, compiling and linking (which is by default a single step), and running. If you encounter problems at some part of this cycle, you may need to refer to the sections that follow this one for help with optimizing, debugging, and so on.

Prerequisite Information:

1. Before you can use the compiler, all the required Linux settings (for example, certain environment variables and storage limits) must be correct for your user ID; for details, see “Correct Settings for Environment Variables” on page 12.
2. To learn more about writing Fortran programs, refer to the *XL Fortran Language Reference*.

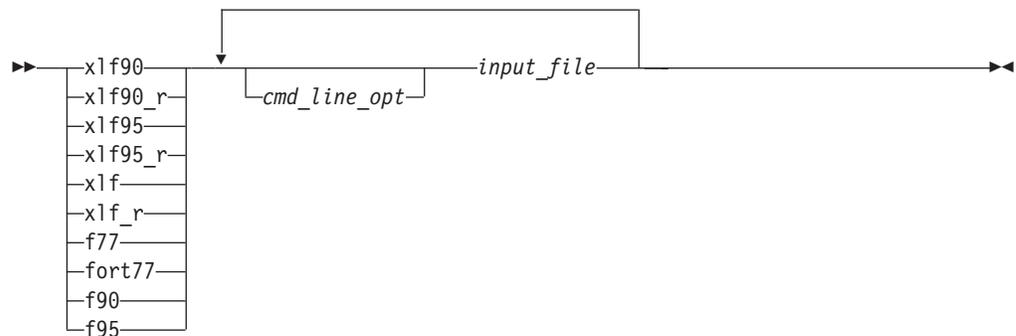
Editing XL Fortran Source Files

To create Fortran source programs, you can use any of the available text editors, such as **vi** or **emacs**. Source programs must have a suffix of **.f** unless the **fsuffix** attribute in the configuration file specifies a different suffix or the **-qsuffix** compiler option is used. You can also use a suffix of **.F** if the program contains C preprocessor (**cpp**) directives that must be processed before compilation begins.

For the Fortran source program to be a valid program, it must conform to the language definition that is specified in the *XL Fortran Language Reference*.

Compiling XL Fortran Programs

To compile a source program, use one of the **xlf90**, **xlf90_r**, **xlf95**, **xlf95_r**, **xlf**, **xlf_r**, **f77**, **fort77**, **f90**, or **f95** commands, which have the form:



These commands all accept essentially the same Fortran language. The main difference is that they use different default options (which you can see by reading the file `/etc/opt/ibmcomp/xlf/9.1/xlf.cfg`).

The invocation command performs the necessary steps to compile the Fortran source files, assemble any **.s** files, and link the object files and libraries into an executable program. In particular, the **xlf_r**, **xlf90_r**, and **xlf95_r** commands use the thread-safe components (libraries, and so on) to link and bind object files.

The following table summarizes the invocation commands that you can use:

Table 1. XL Fortran Invocation Commands

Driver Invocation	Path or Location	Chief Functionality	Linked Libraries
xlf90	/opt/ibmcomp/xlf/9.1/bin	Fortran 90	libxlf90.so
xlf90_r	/opt/ibmcomp/xlf/9.1/bin	Threadsafe Fortran 90	libxlf90_r.so
xlf95	/opt/ibmcomp/xlf/9.1/bin	Fortran 95	libxlf90.so
xlf95_r	/opt/ibmcomp/xlf/9.1/bin	Threadsafe Fortran 95	libxlf90_r.so
xlf	/opt/ibmcomp/xlf/9.1/bin	FORTTRAN 77	libxlf90.so
xlf_r	/opt/ibmcomp/xlf/9.1/bin	Threadsafe FORTRAN 77	libxlf90_r.so
f77 or fort77	/opt/ibmcomp/xlf/9.1/bin	FORTTRAN 77	libxlf90.so
f90	/opt/ibmcomp/xlf/9.1/bin	Fortran 90	libxlf90.so
f95	/opt/ibmcomp/xlf/9.1/bin	Fortran 95	libxlf90.so

The invocation commands have the following implications for directive triggers:

- For **f77**, **fort77**, **f90**, **f95**, **xlf**, **xlf90**, and **xlf95**, the directive trigger is **IBM*** by default.
- For all other commands, the directive triggers are **IBM*** and **IBMT** by default. If you specify **-qsmp**, the compiler also recognizes the **IBMP**, **SMP\$**, and **\$OMP** trigger constants. If you specify the **-qsmp=omp** option, the compiler only recognizes the **\$OMP** trigger constant.

If you specify the **-qsmp** compiler option, the following occurs:

- The compiler turns on automatic parallelization.
- The compiler recognizes the **IBMP**, **IBMT**, **IBM***, **SMP\$**, and **\$OMP** directive triggers.

XL Fortran provides the library **libxlf90_r.so**, in addition to **libxlf90_t.so**. The library **libxlf90_r.so** is a superset of **libxlf90_t.so**. The file **xlf.cfg** is set up to link to **libxlf90_r.so** automatically when you use the **xlf90_r**, **xlf95_r**, and **xlf_r** commands.

libxlf90_t.so is a partial thread-support run-time library. It will be installed as **/opt/ibmcomp/lib/libxlf90_t.so** with one restriction on its use: because routines in the library are not thread-reentrant, only one Fortran thread at a time can perform I/O operations or invoke Fortran intrinsics in a multi-threaded application that uses the library. To avoid the thread synchronization overhead in **libxlf90_r.so**, you can use **libxlf90_t.so** in multi-threaded applications where there is only one Fortran thread.

When you bind a multi-threaded executable with multiple Fortran threads, to link in routines in , **-lxlf90_r** should appear instead of either **-lxlf90_t** or **-lxlf90** in the command line. Note that using the **xlf_r**, **xlf90_r**, or **xlf95_r** invocation command ensures the proper linking.

Compiling Fortran 90 or Fortran 95 Programs

The **f90**, **xlf90**, and **xlf90_r** commands make your programs conform more closely to the Fortran 90 standard than do the **xlf**, **xlf_r**, and **f77/fort77** commands. The **f95**, **xlf95**, and **xlf95_r** commands make your programs conform more closely to the Fortran 95 standard than do the **xlf**, **xlf_r**, and **f77/fort77** commands. **f90**, **xlf90**, **xlf90_r**, **f95**, **xlf95**, and **xlf95_r** are the preferred commands for compiling

any new programs. They all accept Fortran 90 free source form by default; to use them for fixed source form, you must use the **-qfixed** option. I/O formats are slightly different between these commands and the other commands. I/O formats also differ between the set of **xlF90** and **xlF90_r** commands and the set of **xlF95** and **xlF95_r** commands. We recommend that you switch to the Fortran 95 formats for data files whenever possible.

By default, the **xlF90** and **xlF90_r** commands do not conform completely to the Fortran 90 standard. Also, by default, the **xlF95** and **xlF95_r** commands do not conform completely to the Fortran 95 standard. If you need full compliance, compile with any of the following additional compiler options (and suboptions):

```
-qnodirective -qnoescape -qextname -qfloat=nomaf:nofold -qnoswapomp  
-qlanglvl=90std -qlanglvl=95std
```

Also, specify the following run-time options before running the program, with a command similar to the following:

```
export XLFRT_OPTS="err_recovery=no:langlvl=90std"
```

The default settings are intended to provide the best combination of performance and usability. Therefore, it is usually a good idea to change them only when required. Some of the options above are only required for compliance in very specific situations. For example, you only need to specify **-qextname** when an external symbol, such as a common block or subprogram, is named **main**.

Compiling XL Fortran SMP Programs

You can use the **xlF_r**, **xlF90_r**, or **xlF95_r** command to compile XL Fortran SMP programs. The **xlF_r** command is similar to the **xlF** command, the **xlF90_r** command is similar to the **xlF90** command, and the **xlF95_r** command is similar to the **xlF95** command. The main difference is that the thread-safe components are used to link and bind the object files if you specify the **xlF_r**, **xlF90_r**, or **xlF95_r** command.

Note that using any of these commands alone does not imply parallelization. For the compiler to recognize the SMP directives and activate parallelization, you must also specify **-qsmp**. In turn, you can only specify the **-qsmp** option in conjunction with one of these six invocation commands. When you specify **-qsmp**, the driver links in the libraries specified on the **smplibraries** line in the active stanza of the configuration file.

POSIX pthreads API Support

XL Fortran supports thread programming with the IEEE 1003.1-2001 (POSIX) standard pthreads API.

To compile and then link your program with the standard interface libraries, use the **xlF_r**, **xlF90_r**, or **xlF95_r** command. For example, you could specify:

```
xlF95_r test.f
```

Compilation Order for Fortran Programs

If you have a program unit, subprogram, or interface body that uses a module, you must first compile the module. If the module and the code that uses the module are in separate files, you must first compile the file that contains the module. If they are in the same file, the module must come before the code that uses it in the file. If you change any entity in a module, you must recompile any files that use that module.

Canceling a Compilation

To stop the compiler before it finishes compiling, press **Ctrl+C** in interactive mode, or use the **kill** command.

XL Fortran Input Files

The input files to the compiler are:

Source Files (.f or .F suffix)

All **.f** and **.F** files are source files for compilation. When using the **f90** and **f95** invocations, **.f** is not allowed by default; use **.f90** and **.f95**, instead. The compiler compiles source files in the order you specify on the command line. If it cannot find a specified source file, the compiler produces an error message and proceeds to the next file, if one exists. Files with a suffix of **.F** are passed through the C preprocessor (**cpp**) before being compiled.

Include files also contain source and often have different suffixes from **.f**.

Related Information: See “Passing Fortran Files through the C Preprocessor” on page 29.

The **fsuffix** and **cppsuffix** attributes in “Customizing the Configuration File” on page 14 and the “-qsuffix Option” on page 209 let you select a different suffix.

Object Files (.o suffix)

All **.o** files are object files. After the compiler compiles the source files, it uses the **ld** command to link-edit the resulting **.o** files, any **.o** files that you specify as input files, and some of the **.o** and **.a** files in the product and system library directories. It then produces a single executable output file.

Related Information: See “Options That Control Linking” on page 70 and “Linking XL Fortran Programs” on page 32.

The **osuffix** attribute, which is described in “Customizing the Configuration File” on page 14 and the “-qsuffix Option” on page 209, lets you select a different suffix.

Assembler Source Files (.s suffix)

The compiler sends any specified **.s** files to the assembler (**as**). The assembler output consists of object files that are sent to the linker at link time.

Related Information: The **ssuffix** attribute, which is described in “Customizing the Configuration File” on page 14 and the “-qsuffix Option” on page 209, lets you select a different suffix.

Shared Object or Library Files (.so suffix)

These are object files that can be loaded and shared by multiple processes at run time. When a shared object is specified during linking, information about the object is recorded in the output file, but no code from the shared object is actually included in the output file.

Configuration Files (.cfg suffix)

The contents of the configuration file determine many aspects of the compilation process, most commonly the default options for the compiler. You can use it to centralize different sets of default compiler options or to keep multiple levels of the XL Fortran compiler present on a system.

The default configuration file is `/etc/opt/ibmcomp/xlf/9.1/xlf.cfg`.

Related Information: See “Customizing the Configuration File” on page 14 and “-F Option” on page 81 for information about selecting the configuration file.

Module Symbol Files: *modulename.mod*

A module symbol file is an output file from compiling a module and is an input file for subsequent compilations of files that **USE** that module. One **.mod** file is produced for each module, so compiling a single source file may produce multiple **.mod** files.

Related Information: See “-I Option” on page 83 and “-qmoddir Option” on page 168.

Profile Data Files

The **-qpdf1** option produces run-time profile information for use in subsequent compilations. This information is stored in one or more hidden files with names that match the pattern “**.pdf***”.

Related Information: See “-qpdf Option” on page 175.

XL Fortran Output Files

The output files that XL Fortran produces are:

Executable Files: *a.out*

By default, XL Fortran produces an executable file that is named **a.out** in the current directory.

Related Information: See “-o Option” on page 90 for information on selecting a different name and “-c Option” on page 78 for information on generating only an object file.

Object Files: *filename.o*

If you specify the **-c** compiler option, instead of producing an executable file, the compiler produces an object file for each specified **.f** source file, and the assembler produces an object file for each specified **.s** source file. By default, the object files have the same file name prefixes as the source files and appear in the current directory.

Related Information: See “-c Option” on page 78 and “Linking XL Fortran Programs” on page 32. For information on renaming the object file, see “-o Option” on page 90.

Assembler Source Files: *filename.s*

If you specify the **-S** compiler option, instead of producing an executable file, the XL Fortran compiler produces an equivalent assembler source file for each specified **.f** source file. By default, the assembler source files have the same file name prefixes as the source files and appear in the current directory.

Related Information: See “-S Option” on page 233 and “Linking XL Fortran Programs” on page 32. For information on renaming the assembler source file, see “-o Option” on page 90.

Compiler Listing Files: *filename.lst*

By default, no listing is produced unless you specify one or more

listing-related compiler options. The listing file is placed in the current directory, with the same file name prefix as the source file and an extension of `.lst`.

Related Information: See “Options That Control Listings and Messages” on page 61.

Module Symbol Files: `modulename.mod`

Each module has an associated symbol file that holds information needed by program units, subprograms, and interface bodies that **USE** that module. By default, these symbol files must exist in the current directory.

Related Information: For information on putting `.mod` files in a different directory, see “`-qmoddir` Option” on page 168.

cpp-Preprocessed Source Files: `Ffilename.f`

If you specify the `-d` option when compiling a file with a `.F` suffix, the intermediate file created by the C preprocessor (cpp) is saved rather than deleted.

Related Information: See “Passing Fortran Files through the C Preprocessor” on page 29 and “`-d` Option” on page 80.

Profile Data Files (`*.pdf*`)

These are the files that the `-qpdf1` option produces. They are used in subsequent compilations to tune optimizations that are based on actual execution results.

Related Information: See “`-qpdf` Option” on page 175.

Scope and Precedence of Option Settings

You can specify compiler options in any of three locations. Their scope and precedence are defined by the location you use. (XL Fortran also has comment directives, such as **SOURCEFORM**, that can specify option settings. There is no general rule about the scope and precedence of such directives.)

Location	Scope	Precedence
In a stanza of the configuration file.	All compilation units in all files compiled with that stanza in effect.	Lowest
On the command line.	All compilation units in all files compiled with that command.	Medium
In an @PROCESS directive. (XL Fortran also has comment directives, such as SOURCEFORM , that can specify option settings. There is no general rule about the scope and precedence of such directives.)	The following compilation unit.	Highest

If you specify an option more than once with different settings, the last setting generally takes effect. Any exceptions are noted in the individual descriptions in the “Detailed Descriptions of the XL Fortran Compiler Options” on page 73 and are indexed under “conflicting options”.

Specifying Options on the Command Line

XL Fortran supports the traditional UNIX method of specifying command-line options, with one or more letters (known as flags) following a minus sign:

```
xlf95 -c file.f
```

You can often concatenate multiple flags or specify them individually:

```
xlf95 -cv file.f      # These forms
xlf95 -c -v file.f   # are equivalent
```

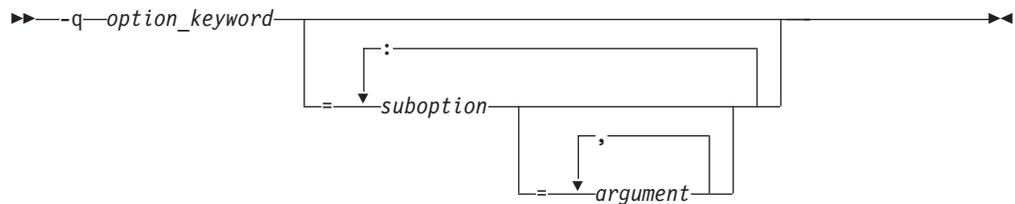
(There are some exceptions, such as **-pg**, which is a single option and not the same as **-p -g**.)

Some of the flags require additional argument strings. Again, XL Fortran is flexible in interpreting them; you can concatenate multiple flags as long as the flag with an argument appears at the end. The following example shows how you can specify flags:

```
# All of these commands are equivalent.
xlf95 -g -v -o montecarlo -p montecarlo.f
xlf95 montecarlo.f -g -v -o montecarlo -p
xlf95 -g -v montecarlo.f -o montecarlo -p
xlf95 -g -v -omontecarlo -p montecarlo.f
# Because -o takes a blank-delimited argument,
# the -p cannot be concatenated.
xlf95 -gvomontecarlo -p montecarlo.f
# Unless we switch the order.
xlf95 -gvpomontecarlo montecarlo.f
```

If you are familiar with other compilers, particularly those in the XL family of compilers, you may already be familiar with many of these flags.

You can also specify many command-line options in a form that is intended to be easy to remember and make compilation scripts and makefiles relatively self-explanatory:



This format is more restrictive about the placement of blanks; you must separate individual **-q** options by blanks, and there must be no blank between a **-q** option and a following argument string. Unlike the names of flag options, **-q** option names are not case-sensitive except that the **q** must be lowercase. Use an equal sign to separate a **-q** option from any arguments it requires, and use colons to separate suboptions within the argument string.

For example:

```
xlf95 -qddim -qXREF=full -qfloat=nomaf:rsqrt -03 -qcache=type=c:level=1 file.f
```

Specifying Options in the Source File

By putting the **@PROCESS** compiler directive in the source file, you can specify compiler options to affect an individual compilation unit. The **@PROCESS** compiler directive can override options specified in the configuration file, in the default settings, or on the command line.



option is the name of a compiler option without the **-q**.

suboption

is a suboption of a compiler option.

In fixed source form, **@PROCESS** can start in column 1 or after column 6. In free source form, the **@PROCESS** compiler directive can start in any column.

You cannot place a statement label or inline comment on the same line as an **@PROCESS** compiler directive.

By default, option settings you designate with the **@PROCESS** compiler directive are effective only for the compilation unit in which the statement appears. If the file has more than one compilation unit, the option setting is reset to its original state before the next unit is compiled. Trigger constants specified by the **DIRECTIVE** option are in effect until the end of the file (or until **NODIRECTIVE** is processed).

The **@PROCESS** compiler directive must usually appear before the first statement of a compilation unit. The only exceptions are when specifying **SOURCE** and **NOSOURCE**; you can put them in **@PROCESS** directives anywhere in the compilation unit.

Passing Command-Line Options to the "ld" or "as" Commands

Because the compiler automatically executes other commands, such as **ld** and **as**, as needed during compilation, you usually do not need to concern yourself with the options of those commands. If you want to choose options for these individual commands, you can do one of the following:

- Include linker options on the compiler command line. When the compiler does not recognize a command-line option other than a **-q** option, it passes the option on to the linker:

```
xlF95 --print-map file.f # --print-map is passed to ld
```

- Use the **-W** compiler option to construct an argument list for the command:

```
xlF95 -Wl,--print-map file.f # --print-map is passed to ld
```

In this example, the **ld** option **--print-map** is passed to the linker (which is denoted by the **I** in the **-Wl** option) when the linker is executed.

This form is more general than the previous one because it works for the **as** command and any other commands called during compilation, by using different letters after the **-W** option.

- Edit the configuration file **/etc/opt/ibmcmp/xlf/9.1/xlf.cfg**, or construct your own configuration file. You can customize particular stanzas to allow specific command-line options to be passed through to the assembler or linker.

For example, if you include these lines in the **xlF95** stanza of **/etc/opt/ibmcmp/xlf/9.1/xlf.cfg**:

```
asopt = "W"
ldopt = "M"
```

and issue this command:

```
xlF95 -Wa,-Z -Wl,-s -w produces_warnings.s uses_many_symbols.f
```

the file **produces_warnings.s** is assembled with the options **-W** and **-Z** (issue warnings and produce an object file even if there are compilation errors), and the linker is invoked with the options **-s** and **-M** (strip final executable file and produce a load map). .

Related Information: See “-W Option” on page 239 and “Customizing the Configuration File” on page 14.

Displaying Information inside Binary Files (strings)

The **strings** command reads information encoded into some binary files, as follows:

- Information about the compiler version is encoded in the compiler binary executables and libraries.
- Information about the parent module, bit mode, the compiler that created the **.mod** file, the date and time the **.mod** file was created, and the source file is encoded in each **.mod** file.

For example to see the information embedded in **/opt/ibmcmp/xlf/9.1/exe/xlfentry**, issue the following command:

```
strings /opt/ibmcmp/xlf/9.1/exe/xlfentry | grep "@(#)"
```

Compiling for Specific Architectures

You can use **-qarch** and **-qtune** to target your program to instruct the compiler to generate code specific to a particular architecture. This allows the compiler to take advantage of machine-specific instructions that can improve performance. The **-qarch** option determines the architectures on which the resulting programs can run. The **-qtune** and **-qcache** options refine the degree of platform-specific optimization performed.

By default, the **-qarch** setting produces code using only instructions common to all supported architectures, with resultant settings of **-qtune** and **-qcache** that are relatively general. To tune performance for a particular processor set or architecture, you may need to specify different settings for one or more of these options. The natural progression to try is to use **-qarch**, and then add **-qtune**, and then add **-qcache**. Because the defaults for **-qarch** also affect the defaults for **-qtune** and **-qcache**, the **-qarch** option is often all that is needed.

If the compiling machine is also the target architecture, **-qarch=auto** will automatically detect the setting for the compiling machine. For more information on this compiler option setting, see “-qarch Option” on page 100 and also **-O4** and **-O5** under the “-O Option” on page 88.

If your programs are intended for execution mostly on particular architectures, you may want to add one or more of these options to the configuration file so that they become the default for all compilations.

Passing Fortran Files through the C Preprocessor

A common programming practice is to pass files through the C preprocessor (**cpp**). **cpp** can include or omit lines from the output file based on user-specified conditions (“conditional compilation”). It can also perform string substitution (“macro expansion”).

XL Fortran can use **cpp** to preprocess a file before compiling it.

To call **cpp** for a particular file, use a file suffix of **.F**. If you specify the **-d** option, each **.F** file *filename.F* is preprocessed into an intermediate file *Ffilename.f*, which is then compiled. If you do not specify the **-d** option, the intermediate file name is */tmpdir/F8xxxxxx*, where *x* is an alphanumeric character and *tmpdir* is the contents of the **TMPDIR** environment variable or, if you have not specified a value for **TMPDIR**, **/tmp**. You can save the intermediate file by specifying the **-d** compiler option; otherwise, the file is deleted. If you only want to preprocess and do not want to produce object or executable files, specify the **-qnoobject** option also.

When XL Fortran uses **cpp** for a file, the preprocessor will emit **#line** directives unless you also specify the **-d** option. The **#line** directive associates code that is created by **cpp** or any other Fortran source code generator with input code that you create. The preprocessor may cause lines of code to be inserted or deleted. Therefore, the **#line** directives that it emits can be useful in error reporting and debugging, because they identify the source statements found in the preprocessed code by listing the line numbers that were used in the original source.

The **_OPENMP C** preprocessor macro can be used to conditionally include code. This macro is defined when the C preprocessor is invoked and when you specify the **-qsmp=omp** compiler option. An example of using this macro follows:

```
program par_mat_mul
  implicit none
  integer(kind=8)                :: i,j,nthreads
  integer(kind=8),parameter      :: N=60
  integer(kind=8),dimension(N,N) :: Ai,Bi,Ci
  integer(kind=8)                :: Sumi
#ifdef _OPENMP
  integer omp_get_num_threads
#endif

  common/data/ Ai,Bi,Ci
!$OMP threadprivate (/data/)

!$omp parallel
  forall(i=1:N,j=1:N) Ai(i,j) = (i-N/2)**2+(j+N/2)
  forall(i=1:N,j=1:N) Bi(i,j) = 3-((i/2)+(j-N/2)**2)
!$omp master
#ifdef _OPENMP
  nthreads=omp_get_num_threads()
#else
  nthreads=8
#endif
!$omp end master
!$omp end parallel

!$OMP parallel default(private),copyin(Ai,Bi),shared(nthreads)
!$omp do
  do i=1,nthreads
    call imat_mul(Sumi)
  enddo
!$omp end do
!$omp end parallel

end
```

See *Conditional Compilation* in the *Language Elements* section of the *XL Fortran Language Reference* for more information on conditional compilation.

To customize **cpp** preprocessing, the configuration file accepts the attributes **cpp**, **cppsuffix**, and **cppoptions**.

The letter F denotes the C preprocessor with the **-t** and **-W** options.

Related Information: See “-d Option” on page 80, “-t Option” on page 234, “-W Option” on page 239, and “Customizing the Configuration File” on page 14.

cpp Directives for XL Fortran Programs

Macro expansion can have unexpected consequences that are difficult to debug, such as modifying a **FORMAT** statement or making a line longer than 72 characters in fixed source form. Therefore, we recommend using **cpp** primarily for conditional compilation of Fortran programs. The **cpp** directives that are most often used for conditional compilation are **#if**, **#ifdef**, **#ifndef**, **#elif**, **#else**, and **#endif**.

Passing Options to the C Preprocessor

Because the compiler does not recognize **cpp** options other than **-I** directly on the command line, you must pass them through the **-W** option. For example, if a program contains **#ifdef** directives that test the existence of a symbol named **LNXXV1**, you can define that symbol to **cpp** by compiling with a command like:

```
xlf95 conditional.F -WF,-DLNXXV1
```

Avoiding Preprocessing Problems

Because Fortran and C differ in their treatment of some sequences of characters, be careful when using **/*** or ***/**. These might be interpreted as C comment delimiters, possibly causing problems even if they occur inside Fortran comments. Also be careful when using three-character sequences that begin with **??** (which might be interpreted as C trigraphs).

Consider the following example:

```
program testcase
character a
character*4 word
a = '?'
word(1:2) = '??'
print *, word(1:2)
end program testcase
```

If the preprocessor matches your character combination with the corresponding trigraph sequence, your output may not be what you expected.

If your code does *not* require the use of the XL Fortran compiler option **-qnoescape**, a possible solution is to replace the character string with an escape sequence **word(1:2) = '\?\?'**. However, if you are using the **-qnoescape** compiler option, this solution will not work. In this case, you require a **cpp** that will ignore the trigraph sequence. XL Fortran uses the **cpp** that is found in **/opt/ibmcmp/xlf/9.1/exe/cpp**. It is **ISO C** compliant and therefore recognizes trigraph sequences.

Linking XL Fortran Programs

By default, you do not need to do anything special to link an XL Fortran program. The compiler invocation commands automatically call the linker to produce an executable output file. For example, running the following command:

```
xlf95 file1.f file2.o file3.f
```

compiles and produces object files `file1.o` and `file3.o`, then all object files are submitted to the linker to produce one executable.

After linking, follow the instructions in “Running XL Fortran Programs” on page 33 to execute the program.

Compiling and Linking in Separate Steps

To produce object files that can be linked later, use the `-c` option.

```
xlf95 -c file1.f           # Produce one object file (file1.o)
xlf95 -c file2.f file3.f   # Or multiple object files (file1.o, file3.o)
xlf95 file1.o file2.o file3.o # Link object files with appropriate libraries
```

It is often best to execute the linker through the compiler invocation command, because it passes some extra `ld` options and library names to the linker automatically.

Passing Options to the `ld` Command

If you need to link with `ld` options that are not part of the XL Fortran default, you can include those options on the compiler command line:

```
xlf95 -Wl,<options...> file.f # xlf95 passes all these options to ld
```

The compiler passes unrecognized options, except `-q` options, to the `ld` command.

Dynamic and Static Linking

XL Fortran allows your programs to take advantage of the operating system facilities for both dynamic and static linking:

- Dynamic linking means that the code for some external routines is located and loaded when the program is first run. When you compile a program that uses shared libraries, the shared libraries are dynamically linked to your program by default.

Dynamically linked programs take up less disk space and less virtual memory if more than one program uses the routines in the shared libraries. During linking, they do not require any special precautions to avoid naming conflicts with library routines. They may perform better than statically linked programs if several programs use the same shared routines at the same time. They also allow you to upgrade the routines in the shared libraries without relinking.

Because this form of linking is the default, you need no additional options to turn it on.

- Static linking means that the code for all routines called by your program becomes part of the executable file.

Statically linked programs can be moved to and run on systems without the XL Fortran libraries. They may perform better than dynamically linked programs if they make many calls to library routines or call many small routines. They do require some precautions in choosing names for data objects and routines in the program if you want to avoid naming conflicts with library routines (as explained in “Avoiding Naming Conflicts during Linking” on page 33). They

also may not work if you compile them on one level of the operating system and run them on a different level of the operating system.

To link statically, add the `--static` option to the linker command. For example:
`xlF95 -Wl,--static test.f`

Avoiding Naming Conflicts during Linking

If you define an external subroutine, external function, or common block with the same name as a run-time subprogram, your definition of that name may be used in its place, or it may cause a link-edit error.

Try the following general solution to help avoid these kinds of naming conflicts:

- Compile all files with the `-qextname` option. It adds an underscore to the end of the name of each global entity, making it distinct from any names in the system libraries.

Note: When you use this option, you do not need to use the final underscore in the names of Service and Utility Subprograms, such as `dtime_` and `flush_`.

- Link your programs dynamically, which is the default.

If you do not use the `-qextname` option, you must take the following extra precautions to avoid conflicts with the names of the external symbols in the XL Fortran and system libraries:

- Do not name a subroutine or function `main`, because XL Fortran defines an entry point `main` to start your program.
- Do not use *any* global names that begin with an underscore. In particular, the XL Fortran libraries reserve all names that begin with `_xl`.
- Do not use names that are the same as names in the XL Fortran library or one of the system libraries. To determine which names are not safe to use in your program, run the `nm` command on any libraries that are linked into the program and search the output for names you suspect might also be in your program.
- If your program calls certain XLF-provided routines, some restrictions apply to the common block and subprogram names that you can use:

XLF-Provided Function Name	Common Block or Subprogram Name You Cannot Use
<code>mclock</code>	<code>times</code>
<code>rand</code>	<code>irand</code>

Be careful not to use the names of subroutines or functions without defining the actual routines in your program. If the name conflicts with a name from one of the libraries, the program could use the wrong version of the routine and not produce any compile-time or link-time errors.

Running XL Fortran Programs

The default file name for the executable program is `a.out`. You can select a different name with the `-o` compiler option. You should avoid giving your programs the same names as system or shell commands (such as `test` or `cp`), as you could accidentally execute the wrong command. If a name conflict does occur, you can execute the program by specifying a path name, such as `./test`.

You can run a program by entering the path name and file name of an executable object file along with any run-time arguments on the command line.

Canceling Execution

To suspend a running program, press the **Ctrl+Z** key while the program is in the foreground. Use the **fg** command to resume running.

To cancel a running program, press the **Ctrl+C** key while the program is in the foreground.

Compiling and Executing on Different Systems

If you want to move an XL Fortran executable file to a different system for execution, you can link statically and copy the program, and optionally the run-time message catalogs. Alternatively, you can link dynamically and copy the program as well as the XL Fortran libraries if needed and optionally the run-time message catalogs. For non-SMP programs, **libxlf90.so** is usually the only XL Fortran library needed. For SMP programs, you will usually need at least the **libxlf90_r.so** and **libxlsmp.so** libraries. **libxlfpm*.so** and **libxlfpad.so** are only needed if the program is compiled with the **-qautodbl** option.

For a dynamically linked program to work correctly, the XL Fortran libraries and the operating system on the execution system must be at either the same level or a more recent level than on the compilation system.

For a statically linked program to work properly, the operating-system level may need to be the same on the execution system as it is on the compilation system.

Related information: See “Dynamic and Static Linking” on page 32.

Run-Time Libraries for POSIX Pthreads Support

There are two run-time libraries that are connected with POSIX thread support. The **libxlf90_r.so** library is a thread-safe version of the Fortran run-time library. The **libxlsmp.so** library is the SMP run-time library.

Depending on the invocation command, and in some cases, the compiler option, the appropriate set of libraries for thread support is bound in. For example:

Cmd.	Libraries Used	Include Directory
xlf90_r	/opt/ibmcomp/lib/libxlf90_r.so	/opt/ibmcomp/xlf/9.1/include
xlf95_r	/opt/ibmcomp/lib64/libxlf90_r.so	
xlf_r	/opt/ibmcomp/lib/libxlsmp.so	
	/opt/ibmcomp/lib64/libxlsmp.so	

Selecting the Language for Run-Time Messages

To select a language for run-time messages that are issued by an XL Fortran program, set the **LANG** and **NLSPATH** environment variables before executing the program.

In addition to setting environment variables, your program should call the C library routine **setlocale** to set the program’s locale at run time. For example, the following program specifies the run-time message category to be set according to the **LC_ALL**, **LC_MESSAGES**, and **LANG** environment variables:

```

PROGRAM MYPROG
PARAMETER(LC_MESSAGES = 5)
EXTERNAL SETLOCALE
CHARACTER NULL_STRING /Z'00'/
CALL SETLOCALE(%VAL(LC_MESSAGES), NULL_STRING)
END

```

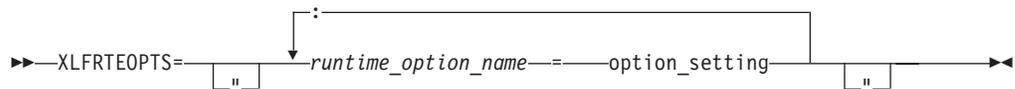
Related Information: See “Environment Variables for National Language Support” on page 12.

Setting Run-Time Options

Internal switches in an XL Fortran program control run-time behavior, similar to the way compiler options control compile-time behavior. You can set the run-time options through either environment variables or a procedure call within the program. You can specify all XL Fortran run-time option settings by using one of two environment variables: **XLFRTEOPTS** and **XLSMPOPTS**.

The XLFRTEOPTS Environment Variable

The **XLFRTEOPTS** environment variable allows you to specify options that affect I/O, EOF error-handling, and the specification of random-number generators. You can declare **XLFRTEOPTS** by using the following **bash** command format:



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 **XLFRTEOPTS** option string contains imbedded blanks, you must enclose the entire option string in double quotation marks (").

The environment variable is checked when the program first encounters one of the following conditions:

- An I/O statement is executed.
- The **RANDOM_SEED** procedure is executed.
- An **ALLOCATE** statement needs to issue a run-time error message.
- A **DEALLOCATE** statement needs to issue a run-time error message.
- The multi-threaded implementation of the **MATMUL** procedure is executed.

Changing the **XLFRTEOPTS** environment variable during the execution of a program has no effect on the program.

The **SETRTEOPTS** procedure (which is defined in the *XL Fortran Language Reference*) accepts a single-string argument that contains the same name-value pairs as the **XLFRTEOPTS** environment variable. It overrides the environment variable and can be used to change settings during the execution of a program. The new settings remain in effect for the rest of the program unless changed by another call to **SETRTEOPTS**. Only the settings that you specified in the procedure call are changed.

You can specify the following run-time options with the **XLFRTEOPTS** environment variable or the **SETRTEOPTS** procedure:

buffering={enable | disable_preconn | disable_all}

Determines whether the XL Fortran run-time library performs buffering for I/O operations.

The library reads data from, or writes data to the file system in chunks for **READ** or **WRITE** statements, instead of piece by piece. The major benefit of buffering is performance improvement.

If you have applications in which Fortran routines work with routines in other languages or in which a Fortran process works with other processes on the same data file, the data written by Fortran routines may not be seen immediately by other parties (and vice versa), because of the buffering. Also, a Fortran **READ** statement may read more data than it needs into the I/O buffer and cause the input operation performed by a routine in other languages or another process that is supposed to read the next data item to fail. In these cases, you can use the **buffering** run-time option to disable the buffering in the XL Fortran run-time library. As a result, a **READ** statement will read in exactly the data it needs from a file and the data written by a **WRITE** statement will be flushed out to the file system at the completion of the statement.

Note: I/O buffering is always enabled for files on sequential access devices (such as pipes, terminals, sockets). The setting of the **buffering** option has no effect on these types of files.

If you disable I/O buffering for a logical unit, you do not need to call the Fortran service routine **flush_** to flush the contents of the I/O buffer for that logical unit.

The suboptions for **buffering** are as follows:

enable	The Fortran run-time library maintains an I/O buffer for each connected logical unit. The current read-write file pointers that the run-time library maintains might not be synchronized with the read-write pointers of the corresponding files in the file system.
disable_preconn	The Fortran run-time library does not maintain an I/O buffer for each preconnected logical unit (0, 5, and 6). However, it does maintain I/O buffers for all other connected logical units. The current read-write file pointers that the run-time library maintains for the preconnected units are the same as the read-write pointers of the corresponding files in the file system.
disable_all	The Fortran run-time library does not maintain I/O buffers for any logical units. You should not specify the buffering=disable_all option with Fortran programs that perform asynchronous I/O.

In the following example, Fortran and C routines read a data file through redirected standard input. First, the main Fortran program reads one integer. Then, the C routine reads one integer. Finally, the main Fortran program reads another integer.

Fortran main program:

```
integer(4) p1,p2,p3
print *,'Reading p1 in Fortran...'
read(5,*) p1
call c_func(p2)
```

```

print *, 'Reading p3 in Fortran...'
read(5,*) p3
print *, 'p1 p2 p3 Read: ', p1, p2, p3
end

```

C subroutine (c_func.c):

```

#include <stdio.h>
void
c_func(int *p2)
{
    int n1 = -1;

    printf("Reading p2 in C...\n");
    setbuf(stdin, NULL); /* Specifies no buffering for stdin */
    fscanf(stdin, "%d", &n1);
    *p2=n1;
    fflush(stdout);
}

```

Input data file (infile):

```

11111
22222
33333
44444

```

The main program runs by using infile as redirected standard input, as follows:

```
$ main < infile
```

If you turn on **buffering=disable_preconn**, the results are as follows:

```

Reading p1 in Fortran...
Reading p2 in C...
Reading p3 in Fortran...
p1 p2 p3 Read: 11111 22222 33333

```

If you turn on **buffering=enable**, the results are unpredictable.

cnvterr={yes | no}

If you set this run-time option to **no**, the program does not obey the **IOSTAT=** and **ERR=** specifiers for I/O statements that encounter conversion errors. Instead, it performs default recovery actions (regardless of the setting of **err_recovery**) and may issue warning messages (depending on the setting of **xrf_messages**).

Related Information: For more information about conversion errors, see *Data Transfer Statements* in the *XL Fortran Language Reference*. For more information about **IOSTAT** values, see *Conditions and IOSTAT Values* in the *XL Fortran Language Reference*.

cpu_time_type={usertime | systemtime | alltime | total_usertime | total_systemtime | total_alltime}

Determines the measure of time returned by a call to **CPU_TIME(TIME)**.

The suboptions for **cpu_time_type** are as follows:

usertime

Returns the user time of a process.

systemtime

Returns the system time of a process.

alltime

Returns the sum of the user and system time of a process.

total_usertime

Returns the total user time of a process. The total user time is the sum of the user time of a process and the total user times of its child processes, if any.

total_systemtime

Returns the total system time of a process. The total system time is the sum of the system time of the current process and the total system times of its child processes, if any.

total_alltime

Returns the total user and system time of a process. The total user and system time is the sum of the user and system time of the current process and the total user and system times of their child processes, if any.

default_recl={64 | 32}

Allows you to determine the default record size for sequential files opened without a **RECL=** specifier. The suboptions are as follows:

64 Uses a 64-bit value as the default record size.

32 Uses a 32-bit value as the default record size.

The **default_recl** run-time option applies only in 64-bit mode. In 32-bit mode, **default_recl** is ignored and the record size is 32-bit.

Use **default_recl** when porting 32-bit programs to 64-bit mode where a 64-bit record length will not fit into the specified integer variable. Consider the following:

```
INTEGER(4) I
OPEN (11)
INQUIRE (11, RECL=i)
```

A run-time error occurs in the above code sample in 64-bit mode when **default_recl=64**, since the default record length of $2^{63}-1$ does not fit into the 4-byte integer I. Specifying **default_recl=32** ensures a default record size of $2^{31}-1$, which fits into I.

For more information on the **RECL=** specifier, see the **OPEN** statement in the *XL Fortran Language Reference*.

erroreof={yes | no}

Determines whether the label specified by the **ERR=** specifier is to be branched to if no **END=** specifier is present when an end-of-file condition is encountered.

err_recovery={yes | no}

If you set this run-time option to **no**, the program stops if there is a recoverable error while executing an I/O statement with no **IOSTAT=** or **ERR=** specifiers. By default, the program takes some recovery action and continues when one of these statements encounters a recoverable error. Setting **cnverr** to **yes** and **err_recovery** to **no** can cause conversion errors to halt the program.

iostat_end={extended | 2003std}

Sets the **IOSTAT** values based on the XL Fortran definition or the Fortran 2003 Draft Standard when end-of-file and end-of-record conditions occur. The suboptions are as follows:

extended

Sets the **IOSTAT** variables based on XL Fortran's definition of values and conditions.

2003std

Sets the **IOSTAT** variables based on Fortran 2003's definition of values and conditions.

For example, setting the **iostat_end=2003std** run-time option results in a different **IOSTAT** value from extensions being returned for the end-of-file condition

```
export XLFRTEOPTS=iostat_end=2003std
character(10) ifl
integer(4) aa(3), ios
ifl = "12344321 "
read(ifl, '(3i4)', iostat=ios) aa ! end-of-file condition occurs and
                                ! ios is set to -1 instead of -2.
```

For more information on setting and using **IOSTAT** values, see the **READ**, **WRITE**, and *Conditions and IOSTAT Values* sections in the *XL Fortran Language Reference*.

intrinths={num_threads}

Specifies the number of threads for parallel execution of the **MATMUL** and **RANDOM_NUMBER** intrinsic procedures. The default value for **num_threads** when using the **MATMUL** intrinsic equals the number of processors online. The default value for **num_threads** when using the **RANDOM_NUMBER** intrinsic is equal to the number of processors online*2.

Changing the number of threads available to the **MATMUL** and **RANDOM_NUMBER** intrinsic procedures can influence performance.

langlvl={extended | 90std | 95std | 2003std}

Determines the level of support for Fortran standards and extensions to the standards. The values of the suboptions are as follows:

- 90std** Specifies that the compiler should flag any extensions to the Fortran 90 standard I/O statements and formats as errors.
- 95std** Specifies that the compiler should flag any extensions to the Fortran 95 standard I/O statements and formats as errors.
- 2003std** Specifies that the compiler should accept all standard I/O statements and formats that the Fortran 95 standard specifies, as well as those Fortran 2003 formats that XL Fortran supports. Anything else is flagged as an error.

For example, setting the **langlvl=2003std** run-time option results in a run-time error message.

```
integer(4) aa(100)
call setrteopts("langlvl=2003std")
...           ! Write to a unit without explicitly
...           ! connecting the unit to a file.
write(10, *) aa ! The implicit connection to a file does not
...           ! conform with Fortran 2003 behavior.
```

extended

Specifies that the compiler should accept the Fortran 95 language standard, Fortran 2003 features supported by XL Fortran, and extensions, effectively turning off language-level checking.

To obtain support for items that are part of the Fortran 95 standard and are available in XL Fortran (such as namelist comments), you must specify one of the following suboptions:

- **95std**
- **2003std**
- **extended**

The following example contains a Fortran 95 extension (the *file* specifier is missing from the **OPEN** statement):

```
program test1

call setrteopts("langlvl=95std")
open(unit=1,access="sequential",form="formatted")

10 format(I3)

write(1,fmt=10) 123

end
```

Specifying **langlvl=95std** results in a run-time error message.

The following example contains a Fortran 95 feature (namelist comments) that was not part of Fortran 90:

```
program test2

INTEGER I
LOGICAL G
NAMELIST /TODAY/G, I

call setrteopts("langlvl=95std:namelist=new")

open(unit=2,file="today.new",form="formatted", &
      & access="sequential", status="old")

read(2,nml=today)
close(2)

end

today.new:

&TODAY ! This is a comment
I = 123, G=.true. /
```

If you specify **langlvl=95std**, no run-time error message is issued. However, if you specify **langlvl=90std**, a run-time error message is issued.

The **err_recovery** setting determines whether any resulting errors are treated as recoverable or severe.

multconn={yes | no}

Enables you to access the same file through more than one logical unit simultaneously. With this option, you can read more than one location within a file simultaneously without making a copy of the file.

You can only use multiple connections within the same program for files on random-access devices, such as disk drives. In particular, you cannot use multiple connections within the same program for:

- Files have been connected for write-only (**ACTION='WRITE'**)
- Asynchronous I/O

- Files on sequential-access devices (such as pipes, terminals, sockets)

To avoid the possibility of damaging the file, keep the following points in mind:

- The second and subsequent **OPEN** statements for the same file can only be for reading.
- If you initially opened the file for both input and output purposes (**ACTION='READWRITE'**), the unit connected to the file by the first **OPEN** becomes read-only (**ACCESS='READ'**) when the second unit is connected. You must close all of the units that are connected to the file and reopen the first unit to restore write access to it.
- Two files are considered to be the same file if they share the same device and i-node numbers. Thus, linked files are considered to be the same file.

multconnio={tty | nulldev | combined | no }

Enables you to connect a device to more than one logical unit. You can then write to, or read from, more than one logical unit that is attached to the same device. The suboptions are as follows:

combined

Enables you to connect a combination of null and TTY devices to more than one logical unit.

nulldev

Enables you to connect the null device to more than one logical unit.

tty Enables you to connect a TTY device to more than one logical unit.

Note: Using this option can produce unpredictable results.

In your program, you can now specify multiple **OPEN** statements that contain different values for the **UNIT** parameters but the same value for the **FILE** parameters. For example, if you have a symbolic link called **mytty** that is linked to TTY device **/dev/tty**, you can run the following program when you specify the **multconnio=tty** option:

```
PROGRAM iotest
OPEN(UNIT=3, FILE='mytty', ACTION="WRITE")
OPEN(UNIT=7, FILE='mytty', ACTION="WRITE")
END PROGRAM iotest
```

Fortran preconnects units 0, 5, and 6 to the same TTY device. Normally, you cannot use the **OPEN** statement to explicitly connect additional units to the TTY device that is connected to units 0, 5, and 6. However, this is possible if you specify the **multconnio=tty** option. For example, if units 0, 5, and 6 are preconnected to TTY device **/dev/tty**, you can run the following program if you specify the **multconnio=tty** option:

```
PROGRAM iotest
! /dev/pts/2 is your current tty, as reported by the 'tty' command.
! (This changes every time you login.)
CALL SETRTEOPTS ('multconnio=tty')
OPEN (UNIT=3, FILE='/dev/pts/2')
WRITE (3, *) 'hello' ! Display 'hello' on your screen
END PROGRAM
```

namelist={new | old}

Determines whether the program uses the XL Fortran new or old **NAMELIST** format for input and output. The Fortran 90 and Fortran 95 standards require the new format.

Note: You may need the **old** setting to read existing data files that contain **NAMELIST** output. However, use the standard-compliant new format in writing any new data files.

With **namelist=old**, the nonstandard **NAMELIST** format is not considered an error by the **langlvl=95std**, **langlvl=90std**, or **langlvl=2003std** setting.

Related Information: For more information about **NAMELIST** I/O, see *Namelist Formatting* in the *XL Fortran Language Reference*.

nlwidth=record_width

By default, a **NAMELIST** write statement produces a single output record long enough to contain all of the written **NAMELIST** items. To restrict **NAMELIST** output records to a given width, use the **nlwidth** run-time option.

Note: The **RECL=** specifier for sequential files has largely made this option obsolete, because programs attempt to fit **NAMELIST** output within the specified record length. You can still use **nlwidth** in conjunction with **RECL=** as long as the **nlwidth** width does not exceed the stated record length for the file.

random={generator1 | generator2}

Specifies the generator to be used by **RANDOM_NUMBER** if **RANDOM_SEED** has not yet been called with the **GENERATOR** argument. The value **generator1** (the default) corresponds to **GENERATOR=1**, and **generator2** corresponds to **GENERATOR=2**. If you call **RANDOM_SEED** with the **GENERATOR** argument, it overrides the random option from that point onward in the program. Changing the random option by calling **SETRTEOPTS** after calling **RANDOM_SEED** with the **GENERATOR** option has no effect.

scratch_vars={yes | no}

To give a specific name to a scratch file, set the **scratch_vars** run-time option to **yes**, and set the environment variable **XLFSCRATCH_unit** to the name of the file you want to be associated with the specified unit number. See "Naming Scratch Files" on page 291 for examples.

unit_vars={yes | no}

To give a specific name to an implicitly connected file or to a file opened with no **FILE=** specifier, you can set the run-time option **unit_vars=yes** and set one or more environment variables with names of the form **XLFUNIT_unit** to file names. See "Naming Files That Are Connected with No Explicit Name" on page 290 for examples.

uwidth={32 | 64}

To specify the width of record length fields in unformatted sequential files, specify the value in bits. When the record length of an unformatted sequential file is greater than $(2^{*31} - 1)$ bytes minus 8 bytes (for the record terminators surrounding the data), you need to set the run-time option **uwidth=64** to extend the record length fields to 64 bits. This allows the record length to be up to $(2^{*63} - 1)$ minus 16 bytes (for the record terminators surrounding the data). The run-time option **uwidth** is only valid for 64-bit mode applications.

xrf_messages={yes | no}

To prevent programs from displaying run-time messages for error conditions during I/O operations, **RANDOM_SEED** calls, and **ALLOCATE** or **DEALLOCATE** statements, set the **xrf_messages** run-time option to **no**. Otherwise, run-time messages for conversion errors and other problems are sent to the standard error stream.

The following examples set the **cnverr** run-time option to **yes** and the **xrf_messages** option to **no**.

```
# Basic format
XLFRTEOPTS=cnverr=yes:xrf_messages=no
export XLFRTEOPTS

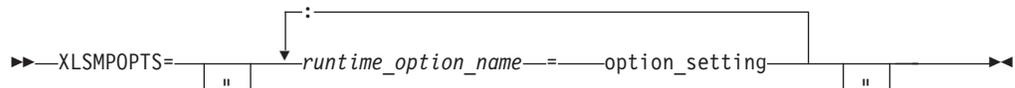
# With imbedded blanks
XLFRTEOPTS="xrf_messages = NO : cnverr = YES"
export XLFRTEOPTS
```

As a call to **SETRTEOPTS**, this example could be:

```
CALL setrteopts('xrf_messages=NO:cnverr=yes')
! Name is in lowercase in case -U (mixed) option is used.
```

The XLSMPOPTS Environment Variable

The **XLSMPOPTS** environment variable allows you to specify options that affect SMP execution. You can declare **XLSMPOPTS** by using the following **bash** command format:



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 (").

You can specify the following run-time options with the **XLSMPOPTS** environment variable:

schedule

Selects the scheduling type and chunk size to be used as the default at run time. The scheduling type that you specify will only be used for loops that were not already marked with a scheduling type at compilation time.

Work is assigned to threads in a different manner, depending on the scheduling type and chunk size used. A brief description of the scheduling types and their influence on how work is assigned follows:

dynamic or guided

The run-time library dynamically schedules parallel work for threads on a "first-come, first-do" basis. "Chunks" of the remaining work are assigned to available threads until all work has been assigned. Work is not assigned to threads that are asleep.

static Chunks of work are assigned to the threads in a "round-robin" fashion. Work is assigned to all threads, both active and asleep. The system must activate sleeping threads in order for them to complete their assigned work.

affinity

The run-time library performs an initial division of the iterations into *number_of_threads* partitions. The number of iterations that these partitions contain is:

$$\text{CEILING}(\text{number_of_iterations} / \text{number_of_threads})$$

These partitions are then assigned to each of the threads. It is these partitions that are then subdivided into chunks of iterations. If a thread is asleep, the threads that are active will complete their assigned partition of work.

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*] As described previously, the iterations of a loop are initially divided into partitions, which are then preassigned to the threads. Each of these partitions is then further subdivided into chunks that contain *n* iterations. If you have not specified *n*, a chunk consists of $\text{CEILING}(\text{number_of_iterations_left_in_local_partition} / 2)$ loop iterations.

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

dynamic[=*n*] The iterations of a loop are divided into chunks that contain *n* iterations each. If you have not specified *n*, a chunk consists of $\text{CEILING}(\text{number_of_iterations} / \text{number_of_threads})$ iterations.

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 you have not specified *n*, the default value for *n* is 1 iteration.

The first chunk contains $\text{CEILING}(\text{number_of_iterations} / \text{number_of_threads})$ iterations. Subsequent chunks consist of $\text{CEILING}(\text{number_of_iterations_left} / \text{number_of_threads})$ iterations.

static[=*n*] The iterations of a loop are divided into chunks that contain *n* iterations. Threads are assigned chunks in a "round-robin" fashion. This is known as block cyclic scheduling. If the value of *n* is 1, the scheduling type is specifically referred to as cyclic scheduling.

If you have not specified *n*, the chunks will contain $\text{CEILING}(\text{number_of_iterations} / \text{number_of_threads})$ iterations. Each thread is assigned one of these chunks. This is known as *block scheduling*.

If you have not specified **schedule**, the default is set to **schedule=static**, resulting in block scheduling.

Related Information: For more information, see the description of the **SCHEDULE** directive in the *XL Fortran Language Reference*.

Parallel execution options

The three parallel execution options, **parthds**, **usrthds**, and **stack**, are as follows:

parthds=*num* Specifies the number of threads (*num*) to be

used for parallel execution of code that you compiled with the **-qsmp** option. By default, this is equal to the number of online processors. There are some applications that cannot use more than some maximum number of processors. There are also some applications that can achieve performance gains if they use more threads than there are processors.

This option allows you full control over the number of execution threads. The default value for *num* is 1 if you did not specify **-qsmp**. Otherwise, it is the number of online processors on the machine. For more information, see the **NUM_PARTHDS** intrinsic function in the *XL Fortran Language Reference*.

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. For more information, see the **NUM_PARTHDS** intrinsic function in the *XL Fortran Language Reference*.

stack=num

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

Set **stack=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.

Performance tuning options

When a thread completes its work and there is no new work to do, it can go into either a "busy-wait" state or a "sleep" state. In "busy-wait", the thread keeps executing in a tight loop looking for additional new work. This state is highly responsive but harms the overall utilization of the system. When a thread sleeps, it completely suspends execution until another thread signals it that there is work to do. This state provides better utilization of the system but introduces extra overhead for the application.

The **xlsmp** run-time library routines use both "busy-wait" and "sleep" states in their approach to waiting for work. You can control these states with the **spins**, **yields**, and **delays** options.

During the busy-wait search for work, the thread repeatedly scans the work queue up to *num* times, where *num* is the value that you specified for the option **spins**. If a thread cannot find work during a given scan, it intentionally wastes cycles in a delay loop that executes *num* times, where *num* is the value that you specified for the option **delays**. This delay loop consists of a single meaningless iteration. The length of actual time this takes will vary among processors. If the value **spins** is exceeded and the thread still cannot find work, the thread will yield the current time slice (time allocated by the processor to that thread) to the other threads. The thread will yield its time slice up to *num* times, where *num* is the number that you specified for the option **yields**. If this value *num* is exceeded, the thread will go to sleep.

In summary, the ordered approach to looking for work consists of the following steps:

1. Scan the work queue for up to **spins** number of times. If no work is found in a scan, then loop **delays** number of times before starting a new scan.
2. If work has not been found, then yield the current time slice.
3. Repeat the above steps up to **yields** number of times.
4. If work has still not been found, then go to sleep.

The syntax for specifying these options is as follows:

spins[=*num*] where *num* is the number of spins before a yield. The default value for **spins** is 100.

yields[=*num*] where *num* is the number of yields before a sleep. The default value for **yields** is 10.

delays[=*num*] where *num* is the number of delays while busy-waiting. The default value for **delays** is 500.

Zero is a special value for **spins** and **yields**, as it can be used to force complete busy-waiting. Normally, in a benchmark test on a dedicated system, you would set both options to zero. However, you can set them individually to achieve other effects.

For instance, on a dedicated 8-way SMP, setting these options to the following:
`parthds=8 : schedule=dynamic=10 : spins=0 : yields=0`

results in one thread per CPU, with each thread assigned chunks consisting of 10 iterations each, with busy-waiting when there is no immediate work to do.

Options to enable and control dynamic profiling

You can use dynamic profiling to reevaluate the compiler's decision to parallelize loops in a program. The three options you can use to do this are: **parthreshold**, **seqthreshold**, and **profilefreq**.

parthreshold=*num* Specifies the time, in milliseconds, below which each loop must execute serially. If you set **parthreshold** 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, **parthreshold** 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.

profilefreq=num

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

Specifies the frequency with which a loop should be revisited by the dynamic profiler to determine its appropriateness for parallel or serial execution. Loops in a program can be data dependent. The loop that was chosen to execute serially with a pass of dynamic profiling may benefit from parallelization in subsequent executions of the loop, due to different data input. Therefore, you need to examine these loops periodically to reevaluate the decision to serialize a parallel loop at run time.

The allowed values for this option are the numbers from 0 to 32. If you set **profilefreq** to one of these values, the following results will occur.

- If **profilefreq** is 0, all profiling is turned off, regardless of other settings. The overheads that occur because of profiling will not be present.
- If **profilefreq** is 1, loops parallelized automatically by the compiler will be monitored every time they are executed.
- If **profilefreq** is 2, loops parallelized automatically by the compiler will be monitored every other time they are executed.
- If **profilefreq** is greater than or equal to 2 but less than or equal to 32, each loop will be monitored once every *n*th time it is executed.
- If **profilefreq** is greater than 32, then 32 is assumed.

It is important to note that dynamic profiling is not applicable to user-specified parallel loops (for example, loops for which you specified the **PARALLEL DO** directive).

OpenMP Environment Variables

The following environment variables, which are included in the OpenMP standard, allow you to control the execution of parallel code.

Note: If you specify both the **XLSMPOPTS** environment variable and an OpenMP environment variable, the OpenMP environment variable takes precedence.

OMP_DYNAMIC Environment Variable

The **OMP_DYNAMIC** environment variable enables or disables dynamic adjustment of the number of threads available for the execution of parallel regions. The syntax is as follows:

▶▶—OMP_DYNAMIC=—

TRUE
FALSE

————▶▶

If you set this environment variable to **TRUE**, the run-time environment can adjust the number of threads it uses for executing parallel regions so that it makes the most efficient use of system resources. If you set this environment variable to **FALSE**, dynamic adjustment is disabled.

The default value for **OMP_DYNAMIC** is **TRUE**. Therefore, if your code needs to use a specific number of threads to run correctly, you should disable dynamic thread adjustment.

The **omp_set_dynamic** subroutine takes precedence over the **OMP_DYNAMIC** environment variable.

OMP_NESTED Environment Variable

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

▶▶—OMP_NESTED=—

TRUE
FALSE

————▶▶

If you set this environment variable to **TRUE**, nested parallelism is enabled. This means that the run-time 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.

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

The **omp_set_nested** subroutine takes precedence over the **OMP_NESTED** environment variable.

OMP_NUM_THREADS Environment Variable

The **OMP_NUM_THREADS** environment variable sets the number of threads that a program will use when it runs. The syntax is as follows:

▶▶—OMP_NUM_THREADS=*num*————▶▶

num the maximum number of threads that can be used if dynamic adjustment of the number of threads is enabled. If dynamic adjustment of the number of threads is not enabled, the value of **OMP_NUM_THREADS** is the exact number of threads that can be used. It must be a positive, scalar integer.

The default number of threads that a program uses when it runs is the number of online processors on the machine.

If you specify the number of threads with both the **PARTHDS** suboption of the **XLSMPOPTS** environment variable and the **OMP_NUM_THREADS** environment variable, the **OMP_NUM_THREADS** environment variable takes precedence. The **omp_set_num_threads** subroutine takes precedence over the **OMP_NUM_THREADS** environment variable.

The following example shows how you can set the **OMP_NUM_THREADS** environment variable:

```
export OMP_NUM_THREADS=16
```

OMP_SCHEDULE Environment Variable

The **OMP_SCHEDULE** environment variable applies to **PARALLEL DO** and work-sharing **DO** directives that have a schedule type of **RUNTIME**. The syntax is as follows:

```
OMP_SCHEDULE=—sched_type— [—, —chunk_size—]
```

sched_type

is either **DYNAMIC**, **GUIDED**, or **STATIC**.

chunk_size

is a positive, scalar integer that represents the chunk size.

This environment variable is ignored for **PARALLEL DO** and work-sharing **DO** directives that have a schedule type other than **RUNTIME**.

If you have not specified a schedule type either at compile time (through a directive) or at run time (through the **OMP_SCHEDULE** environment variable or the **SCHEDULE** option of the **XLSMPOPTS** environment variable), the default schedule type is **STATIC**, and the default chunk size is set to the following for the first $N - 1$ threads:

```
chunk_size = ceiling(Iter/N)
```

It is set to the following for the N th thread, where N is the total number of threads and $Iter$ is the total number of iterations in the **DO** loop:

```
chunk_size = Iter - ((N - 1) * ceiling(Iter/N))
```

If you specify both the **SCHEDULE** option of the **XLSMPOPTS** environment variable and the **OMP_SCHEDULE** environment variable, the **OMP_SCHEDULE** environment variable takes precedence.

The following examples show how you can set the **OMP_SCHEDULE** environment variable:

```
export OMP_SCHEDULE="GUIDED,4"  
export OMP_SCHEDULE="DYNAMIC"
```

Other Environment Variables That Affect Run-Time Behavior

The **LD_LIBRARY_PATH**, **LD_RUN_PATH**, and **TMPDIR** environment variables have an effect at run time, as explained in “Correct Settings for Environment Variables” on page 12. They are not XL Fortran run-time options and cannot be set in either **XLFRTEOPTS** or **XLSMPOPTS**.

XL Fortran Run-Time Exceptions

The following operations cause run-time exceptions in the form of **SIGTRAP** signals, which typically result in a “Trace/BPT trap” message:

- Character substring expression or array subscript out of bounds after you specified the **-C** option at compile time.
- Lengths of character pointer and target do not match after you specified the **-C** option at compile time.
- The flow of control in the program reaches a location for which a semantic error with severity of **S** was issued when the program was compiled.

- Floating-point operations that generate NaN values and loads of the NaN values after you specify the **-qfloat=nanq** option at compile time.
- Fixed-point division by zero.

The following operations cause run-time exceptions in the form of **SIGFPE** signals:

- Floating-point exceptions provided you specify the appropriate **-qflttrap** suboptions at compile time.

If you install one of the predefined XL Fortran exception handlers before the exception occurs, a diagnostic message and a traceback showing the offset within each routine called that led to the exception are written to standard error after the exception occurs. The file buffers are also flushed before the program ends. If you compile the program with the **-g** option, the traceback shows source line numbers in addition to the address offsets.

You can use a symbolic debugger to determine the error. **gdb** provides a specific error message that describes the cause of the exception.

Related Information: See “-C Option” on page 77, “-qflttrap Option” on page 135, and “-qsigtrap Option” on page 197.

See “Detecting and Trapping Floating-Point Exceptions” on page 257 for more details about these exceptions and “Controlling the Floating-Point Status and Control Register” on page 260 for a list of exception handlers.

XL Fortran Compiler-Option Reference

This section contains the following:

- Tables of compiler options. These tables are organized according to area of use and contain high-level information about the syntax and purpose of each option.
- Detailed information about each compiler option in “Detailed Descriptions of the XL Fortran Compiler Options” on page 73.

Summary of the XL Fortran Compiler Options

The following tables show the compiler options available in the XL Fortran compiler that you can enter in the configuration file, on the command line, or in the Fortran source code by using the **@PROCESS** compiler directive.

You can enter compiler options that start with **-q**, suboptions, and **@PROCESS** directives in either uppercase or lowercase. However, note that if you specify the **-qmixed** option, procedure names that you specify for the **-qextern** option are case-sensitive.

In general, this document uses the convention of lowercase for **-q** compiler options and suboptions and uppercase for **@PROCESS** directives. However, in the “Syntax” sections of this section and in the “Command-Line Option” column of the summary tables, we use uppercase letters in the names of **-q** options, suboptions, and **@PROCESS** directives to represent the minimum abbreviation for the keyword. For example, valid abbreviations for **-qOPTimize** are **-qopt**, **-qopti**, and so on.

Understanding the significance of the options you use and knowing the alternatives available can save you considerable time and effort in making your programs work correctly and efficiently.

Options That Control Input to the Compiler

The following options affect the compiler input at a high level. They determine which source files are processed and select case sensitivity, column sensitivity, and other global format issues.

Related Information: See “XL Fortran Input Files” on page 24 and “Options That Specify the Locations of Output Files” on page 54.

Many of the options in “Options for Compatibility” on page 62 change the permitted input format slightly.

Table 2. Options That Control Input to the Compiler

Command-Line Option	@PROCESS Directive	Description	See Page
-I <i>dir</i>		<p>Adds a directory to the search path for include files and .mod files. If XL Fortran calls cpp, this option adds a directory to the search path for #include files. Before checking the default directories for include and .mod files, the compiler checks each directory in the search path. For include files, this path is only used if the file name in an INCLUDE line is not provided with an absolute path. For #include files, refer to the cpp documentation for the details of the -I option.</p> <p>Default: The following directories are searched, in the following order:</p> <ol style="list-style-type: none"> 1. The current directory 2. The directory where the source file is 3. /usr/include. <p>Also, /opt/ibmcomp/xf9.1/include is searched; include and .mod files that are shipped with the compiler are located here.</p>	83
-qci= <i>numbers</i>	CI(<i>numbers</i>)	<p>Activates the specified INCLUDE lines.</p> <p>Default: No default value.</p>	113
-qcr -qnocr		<p>Allows you to control how the compiler interprets the CR (carriage return) character. This allows you to compile code written using a Mac OS or DOS/Windows editor.</p> <p>Default: -qcr</p>	115

Table 2. Options That Control Input to the Compiler (continued)

Command-Line Option	@PROCESS Directive	Description	See Page
-qdirective [= <i>directive_list</i>]	DIRECTIVE [[<i>directive_list</i>]]	Specifies sequences of characters, known as trigger constants, that identify comment lines as compiler comment directives. Default: Comment lines beginning with IBM* are considered directives. If you specify -qsmp=omp , only \$OMP is considered to be a directive trigger. All other directive triggers are turned off unless you explicitly turn them back on. If you specify -qsmp=noomp (noomp is the default for -qsmp), IBMP , \$OMP and SMP\$ are considered directive triggers, along with any other directive triggers that are turned on (such as IBM* and IBMT). If you have also specified -qthreaded , comment lines beginning with IBMT are also considered directives.	120
-qnodirective [= <i>directive_list</i>]	NODIRECTIVE [[<i>directive_list</i>]]		
-qfixed [= <i>right_margin</i>]	FIXED [[<i>right_margin</i>]]	Indicates that the input source program is in fixed source form and optionally specifies the maximum line length. Default: -qfree=f90 for the f90 , f95 , xl f90 , xl f90_r , xl f95 , and xl f95_r commands and -qfixed=72 for the xl f , xl f_r , and f77/fort77 commands.	131
-qfree[={f90 ibm}] -k	FREE[({F90 IBM})]	Indicates that the source code is in free form. The ibm and f90 suboptions specify compatibility with the free source form defined for VS FORTRAN and Fortran 90/Fortran 95, respectively. -k and -qfree are short forms for -qfree=f90 . Default: -qfree=f90 for the f90 , f95 , xl f90 , xl f90_r , xl f95 , and xl f95_r commands and -qfixed=72 for the xl f , xl f_r , and f77/fort77 commands.	137
-qmbsc -qnombsc	MBCS NOMBCS	Indicates to the compiler whether character literal constants, Hollerith constants, H edit descriptors, and character string edit descriptors can contain Multibyte Character Set (MBCS) or Unicode characters. Default: -qnombsc	165
-U -qmixed -qnomixed	MIXED NOMIXED	Makes the compiler sensitive to the case of letters in names. Default: -qnomixed	235

Table 2. Options That Control Input to the Compiler (continued)

Command-Line Option	@PROCESS Directive	Description	See Page
-qsuffix={suboptions}		Specifies the source-file suffix on the command line.	209

Options That Specify the Locations of Output Files

The following options specify names or directories where the compiler stores output files.

In the table, an * indicates that the option is processed by the `ld` command, rather than by the XL Fortran compiler; you can find more information about these options in the Linux information for the `ld` command.

Related Information: See “XL Fortran Output Files” on page 25 and “Options That Control Input to the Compiler” on page 52.

Table 3. Options That Specify the Locations of Output Files

Command-Line Option	@PROCESS Directive	Description	See Page
-d		Leaves preprocessed source files produced by <code>cpp</code> , instead of deleting them. Default: Temporary files produced by <code>cpp</code> are deleted.	80
-o <i>name</i> *		Specifies a name for the output object, executable, or assembler source file. Default: <code>-o a.out</code>	90
-qmoddir= <i>directory</i>		Specifies the location for any module (<code>.mod</code>) files that the compiler writes. Default: <code>.mod</code> files are placed in the current directory.	168

Options for Performance Optimization

The following options can help you to speed up the execution of your XL Fortran programs or to find areas of poor performance that can then be tuned. The most important such option is `-O`. In general, the other performance-related options work much better in combination with `-O`; some have no effect at all without `-O`.

Related Information: See “Optimizing XL Fortran Programs” on page 265.

Some of the options in “Options for Floating-Point Processing” on page 70 can also improve performance, but you must use them with care to avoid error conditions and incorrect results.

Table 4. Options for Performance Optimization

Command-Line Option	@PROCESS Directive	Description	See Page
-O[<i>level</i>] -qoptimize[= <i>level</i>] -qnooptimize	OPTimize[<i>level</i>] NOOPTimize	Specifies whether code is optimized during compilation and, if so, at which level (0, 2, 3, 4, or 5). Default: <code>-qnooptimize</code>	88

Table 4. Options for Performance Optimization (continued)

Command-Line Option	@PROCESS Directive	Description	See Page
-p -pg		Sets up the object file for profiling. Default: No profiling.	91
-Q -Q! -Q+ <i>names</i> -Q- <i>names</i>		Specifies whether procedures are inlined and/or particular procedures that should or should not be inlined. <i>names</i> is a list of procedure names separated by colons. Default: No inlining.	92
-qalias= {[no]aryovrlp [no]intptr [no]pteovrlp [no]std...}	ALIAS({[NO]ARYOVRLP [NO]INTPTR [NO]PTEOVRLP [NO]STD...)	Indicates whether a program contains certain categories of aliasing. The compiler limits the scope of some optimizations when there is a possibility that different names are aliases for the same storage locations. Default: - qalias=aryovrlp:nointptr:pteovrlp:std for the xlF90 , xlF90_r , xlF95 , xlF95_r , F90 , and F95 commands; -qalias=aryovrlp:intptr:pteovrlp:std for the xlF , xlF_r , F77 , and fort77 , commands.	95
-qalign={[no]4k struct {=subopt} bindc {=subopt}}	ALIGN({[NO]4K STRUCT({subopt}) BINDC({subopt}))	Specifies the alignment of data objects in storage, which avoids performance problems with misaligned data. The [no]4k , bindc , and struct options can be specified and are not mutually exclusive. The [no]4k option is useful primarily in combination with logical volume I/O and disk striping. Default: - qalign=no4k:struct=natural:bindc=linuxppc	98
-qarch= <i>architecture</i>		Controls which instructions the compiler can generate. Changing the default can improve performance but might produce code that can only be run on specific machines. For Y-HPC, only auto and ppc970 are valid. Otherwise, the choices are com, ppc, ppcgr, ppc64, ppc64gr, ppc64grsq, rs64b, rs64c, pwr3, pwr4, pwr5, and ppc970. Default: For Y-HPC, -qarch=ppc970 ; for all others, -qarch=ppc64grsq	100
-qassert={ deps nodeps itercnt= <i>n</i> }		Provides information about the characteristics of the files that can help to fine-tune optimizations. Default: -qassert=deps:itercnt=1024	104
-qcache={ auto assoc= <i>number</i> cost= <i>cycles</i> level= <i>level</i> line= <i>bytes</i> size= <i>Kbytes</i> type={C c D d I i}}[:...]		Specifies the cache configuration for a specific execution machine. The compiler uses this information to tune program performance, especially for loop operations that can be structured (or <i>blocked</i>) to process only the amount of data that can fit into the data cache. Default: The compiler uses typical values based on the -qtune setting, the -qarch setting, or both settings.	109

Table 4. Options for Performance Optimization (continued)

Command-Line Option	@PROCESS Directive	Description	See Page
-qcompact -qnocompact	COMPACT NOCOMPACT	Reduces optimizations that increase code size. Default: -qnocompact	114
-qdirectstorage -qnodirectstorage		Informs the compiler that a given compilation unit may reference write-through-enabled or cache-inhibited storage. 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. Default: -qnodirectstorage	122
-qenablevmx -qnoenablevmx		Instructs the compiler to generate VMX (Vector Multimedia eXtension) instructions. The -qhot=simd and -qarch=ppc970 compiler options must be specified, as well. Default: -qnoenablevmx	125
-qessl		Allows the use of ESSL routines in place of Fortran 90 Intrinsic Procedures. Use the ESSL Serial Library when linking with -lessl . Use the ESSL SMP Library when linking with -lesslsmpl . Default: -qnoessl	127
-qhot[= <i>suboptions</i>] -qnohot	HOT[= <i>suboptions</i>] NOHOT	The -qhot compiler option is a powerful alternative to hand tuning that provides opportunities to optimize loops and array language. The -qhot compiler option will always attempt to optimize loops, regardless of the suboptions you specify. Default: -qnohot	140
-qipa[= <i>suboptions</i>] -qnoipa		Enhances -O optimization by doing detailed analysis across procedures (interprocedural analysis or IPA). Default: -O analyzes each subprogram separately, ruling out certain optimizations that apply across subprogram boundaries. Note that specifying -O5 is equivalent to specifying -O4 and -qipa=level=2 .	149
-qkeepparm -qnokeepparm		Ensures that incoming procedure parameters are stored on the stack even when optimizing. Default: -qnokeepparm	155
-qmaxmem= Kbytes	MAXMEM (Kbytes)	Limits the amount of memory that the compiler allocates while performing specific, memory-intensive optimizations to the specified number of kilobytes. A value of -1 allows optimization to take as much memory as it needs without checking for limits. Default: -qmaxmem=2048; At -O3, -O4, and -O5, -qmaxmem=-1.	163

Table 4. Options for Performance Optimization (continued)

Command-Line Option	@PROCESS Directive	Description	See Page
-qObject -qNOObject	Object NOObject	Specifies whether to produce an object file or to stop immediately after checking the syntax of the source files. Default: -qobject	172
-qpdf{1 2}		Tunes optimizations through <i>profile-directed feedback</i> (PDF), where results from sample program execution are used to improve optimization near conditional branches and in frequently executed code sections. Default: Optimizations use fixed assumptions about branch frequency and other statistics.	175
-qprefetch -qnoprefetch		Indicates whether or not the prefetch instructions should be inserted automatically by the compiler. Default: -qprefetch	185
-qsaveopt -qnosaveopt		Saves the command-line options used for compiling a source file in the corresponding object file. Default: -qnosaveopt	194
-qshowpdf -qnoshowpdf		Adds additional call and block count profiling information to an executable. This option is used together with the -qpdf1 option. Default: -qnoshowpdf	196
-qsmallstack[= <i>dynlenonheap</i>] -qnosmallstack		Specifies that the compiler will minimize stack usage where possible. Default: -qnosmallstack	198
-qsmpt[= <i>suboptions</i>] -qnosmp		When used with xlfr_r , xlfr90_r , or xlfr95_r , controls automatic parallelization of loops, user-directed parallelization of loops and other items, and the choice of chunking algorithm. Default: -qnosmp	199
-NSbytes -qSPILLsize= bytes	SPILLsize (bytes)	Specifies the size of internal program storage areas. Default: -NS512	87
-qstrict -qnostrict	STRICT NOSTRICT	Ensures that optimizations done by the -O3 , -O4 , -O5 , -qhot , and -qipa options do not alter the semantics of a program. Default: With -O3 and higher levels of optimization in effect, code may be rearranged so that results or exceptions are different from those in unoptimized programs. For -O2 , the default is -qstrict . This option is ignored for -qnoopt .	206
-qstrictieemod -qnostrictieemod	STRICTIEEE- MOD NOSTRICTIEEE- MOD	Specifies whether the compiler will adhere to the Fortran 2003 IEEE arithmetic rules for the ieee_arithmetic and ieee_exceptions intrinsic modules. Default: -qstrictieemod	207

Table 4. Options for Performance Optimization (continued)

Command-Line Option	@PROCESS Directive	Description	See Page
-qstrict_induction -qnostrict_induction		Prevents the compiler from performing induction (loop counter) variable optimizations. These optimizations may be <i>unsafe</i> (may alter the semantics of your program) when there are integer overflow operations involving the induction variables. Default: -qnostrict_induction	208
-qthreaded		Specifies that the compiler should generate thread-safe code. This is turned on by default for the <code>xlf_r</code> , <code>xlf90_r</code> , and <code>xlf95_r</code> commands.	215
-qtune= <i>implementation</i>		Tunes instruction selection, scheduling, and other implementation-dependent performance enhancements for a specific implementation of a hardware architecture. For Y-HPC, only <code>auto</code> and <code>ppc970</code> are valid. For all others, the following settings are valid: <code>auto</code> , <code>rs64b</code> , <code>rs64c</code> , <code>pwr3</code> , <code>pwr4</code> , <code>pwr5</code> , or <code>ppc970</code> . Default: For Y-HPC, -qtune=ppc970; for all others, -qtune=pwr4	216
-qunroll [=auto yes] -qnounroll		Specifies whether the compiler is allowed to automatically unroll DO loops. Default: -qunroll=auto	219
-qunwind -qnounwind	UNWIND NOUNWIND	Specifies default behavior for saving and restoring from non-volatile registers during a procedure call. Default: -qunwind	220
-qversion -qnoverion		Displays the version and release of the invoking compiler. Default: -qnoverion	221
-qzerosize -qnozerosize	ZEROSIZE NOZEROSIZE	Improves performance of FORTRAN 77 and some Fortran 90 and Fortran 95 programs by preventing checking for zero-sized character strings and arrays. Default: -qzerosize for the xlf90, xlf90_r, xlf95, xlf95_r, f90, and f95 commands and -qnozerosize for the xlf, xlf_r, f77, and fort77 commands (meaning these commands cannot be used for programs that contain zero-sized objects).	232

Options for Error Checking and Debugging

The following options help you avoid, detect, and correct problems in your XL Fortran programs and can save you having to refer as frequently to “Problem Determination and Debugging” on page 325.

In particular, **-qlanglvl** helps detect portability problems early in the compilation process by warning of potential violations of the Fortran standards. These can be due to extensions in the program or due to compiler options that allow such extensions.

Other options, such as **-C** and **-qfltrap**, detect and/or prevent run-time errors in calculations, which could otherwise produce incorrect output.

Because these options require additional checking at compile time and some of them introduce run-time error checking that slows execution, you may need to experiment to find the right balance between extra checking and slower compilation and execution performance.

Using these options can help to minimize the amount of problem determination and debugging you have to do. Other options you may find useful while debugging include:

- “# Option” on page 74, “-v Option” on page 237, and “-V Option” on page 238
- “-qalias Option” on page 95
- “-qci Option” on page 113
- “-qobject Option” on page 172
- “-qreport Option” on page 190
- “-qsource Option” on page 204

Table 5. Options for Debugging and Error Checking

Command-Line Option	@PROCESS Directive	Description	See Page
-C -qcheck -qnocheck	CHECK NOCHECK	Checks each reference to an array element, array section, or character substring for correctness. Out-of-bounds references are reported as severe errors if found at compile time and generate SIGTRAP signals at run time. Default: -qnocheck	77
-D -qdlines -qnodlines	DLINES NODLINES	Specifies whether fixed source form lines with a D in column 1 are compiled or treated as comments. Default: -qnodlines	79
-g -qdbg -qnodbg	DBG NODBG	Generates debug information for use by a symbolic debugger. Default: -qnodbg	82
-qfltrap [= <i>suboptions</i>] -qnofltrap	FLTRAP [[<i>suboptions</i>]] NOFLTRAP	Determines what types of floating-point exception conditions to detect at run time. The program receives a SIGFPE signal when the corresponding exception occurs. Default: -qnofltrap	135
-qfullpath -qnofullpath		Records the full, or absolute, path names of source and include files in object files compiled with debugging information (-g option). Default: The relative path names of source files are recorded in the object files.	138
-qhalt= <i>sev</i>	HALT(<i>sev</i>)	Stops before producing any object, executable, or assembler source files if the maximum severity of compile-time messages equals or exceeds the specified severity. <i>severity</i> is one of i, l, w, e, s, u, or q, meaning informational, language, warning, error, severe error, unrecoverable error, or a severity indicating “don’t stop”. Default: -qhalt=S	139

Table 5. Options for Debugging and Error Checking (continued)

Command-Line Option	@PROCESS Directive	Description	See Page
-qinitauto[= <i>hex_value</i>] -qnoinitauto		Initializes each byte or word (4 bytes) of storage for automatic variables to a specific value, depending on the length of the <i>hex_value</i> . This helps you to locate variables that are referenced before being defined. For example, by using both the -qinitauto option to initialize REAL variables with a signaling NAN value and the -qflttrap option, it is possible to identify references to uninitialized REAL variables at run time. Default: -qnoinitauto . If you specify -qinitauto without a <i>hex_value</i> , the compiler initializes the value of each byte of automatic storage to zero.	144
-qlanglvl={ 77std 90std 90pure 95std 95pure extended}	LANGLVL({ 77STD 90STD 90PURE 95STD 95PURE EXTENDED})	Determines which language standard (or superset, or subset of a standard) to consult for nonconformance. It identifies nonconforming source code and also options that allow such nonconformances. Default: -qlanglvl=extended	156
-qsaa -qnosaa	SAA NOSAA	Checks for conformance to the SAA FORTRAN language definition. It identifies nonconforming source code and also options that allow such nonconformances. Default: -qnosaa	192
-qsigtrap[= <i>trap_handler</i>]		Installs <code>xl__trce</code> or a predefined or user-written trap handler in a main program. Default: No trap handler installed; program core dumps when a trap instruction is executed.	197
-qtbtable={none small full}		Only applies to the 64-bit environment. Limits the amount of debugging traceback information in object files, to reduce the size of the program. Default: Full traceback information in the object file when compiling non-optimized (without -O) or for debugging (with -g). Otherwise, a small amount of traceback information in the object file.	214
-qxflag=dvz		Specifying -qxflag=dvz causes the compiler to generate code to detect floating-point divide-by-zero operations. Default: No code is generated to detect floating-point divide-by-zero operations.	223
-qxlines -qnoxlines	XLINES NOXLINES	Specifies whether fixed source form lines with a X in column 1 are treated as source code and compiled, or treated instead as comments. Default: -qnoxlines	229

Options That Control Listings and Messages

The following options determine whether the compiler produces a listing (.lst file), what kinds of information go into the listing, and what the compiler does about any error conditions it detects.

Some of the options in “Options for Error Checking and Debugging” on page 58 can also produce compiler messages.

Table 6. Options That Control Listings and Messages

Command-Line Option	@PROCESS Directive	Description	See Page
-#		Generates information on the progress of the compilation without actually running the individual components. Default: No progress messages are produced.	74
-qattr[=full] -qnoattr	ATTR[FULL] NOATTR	Specifies whether to produce the attribute component of the attribute and cross-reference section of the listing. Default: -qnoattr	105
-qflag= <i>listing_severity</i> : <i>terminal_severity</i> -w	FLAG (<i>listing_severity</i> , <i>terminal_severity</i>)	Limits the diagnostic messages to those of a specified level or higher. Only messages with severity <i>listing_severity</i> or higher are written to the listing file. Only messages with severity <i>terminal_severity</i> or higher are written to the terminal. -w is a short form for -qflag=e:e. Default: -qflag=i:i	132
-qlist -qnolist	LIST NOLIST	Specifies whether to produce the object section of the listing. Default: -qnolist	160
-qlistopt -qnolistopt	LISTOPT NOLISTOPT	Determines whether to show the setting of every compiler option in the listing file or only selected options. These selected options include those specified on the command line or directives plus some that are always put in the listing. Default: -qnolistopt	161
-qnoprint		Prevents the listing file from being created, regardless of the settings of other listing options. Default: Listing is produced if you specify any of -qattr, -qlist, -qlistopt, -qphsinfo, -qreport, -qsource, or -qxref.	170
-qphsinfo -qnophsinfo	PHSINFO NOPHSINFO	Determines whether timing information is displayed on the terminal for each compiler phase. Default: -qnophsinfo	179
-qreport[={simplist hotlist}...] -qnoreport	REPORT [({SMPLIST HOTLIST}...)] NOREPORT	Determines whether to produce transformation reports showing how the program is parallelized and how loops are optimized. Default: -qnoreport	190

Table 6. Options That Control Listings and Messages (continued)

Command-Line Option	@PROCESS Directive	Description	See Page
-qsource -qnosource	SOURCE NOSOURCE	Determines whether to produce the source section of the listing. Default: -qnosource	204
-qsuppress [= <i>nnnn-mmm[:nnnn-mmm...]</i> cmpmsg] -qnosuppress		Specifies which messages to suppress from the output stream.	210
-qxref -qnoxref -qxref=full	XREF NOXREF XREF(FULL)	Determines whether to produce the cross-reference component of the attribute and cross-reference section of the listing. Default: -qnoxref	231
-S		Produces one or more .s files showing equivalent assembler source for each Fortran source file. Default: No equivalent assembler source is produced.	233
-v		Traces the progress of the compilation by displaying the name and parameters of each compiler component that is executed by the invocation command. Default: No progress messages are produced.	237
-V		Traces the progress of the compilation by displaying the name and parameters of each compiler component that is executed by the invocation command. These are displayed in a shell-executable format. Default: No progress messages are produced.	238

Options for Compatibility

The following options help you maintain compatibility between your XL Fortran source code on past, current, and future hardware platforms or help you port programs to XL Fortran with a minimum of changes.

Related Information: “Porting Programs to XL Fortran” on page 339 discusses this subject in more detail. “Duplicating the Floating-Point Results of Other Systems” on page 257 explains how to use some of the options in “Options for Floating-Point Processing” on page 70 to achieve floating-point results compatible with other systems.

The **-qfree=ibm** form of the “-qfree Option” on page 137 also provides compatibility with VS FORTRAN free source form.

Table 7. Options for Compatibility

Command-Line Option	@PROCESS Directive	Description	See Page
-qautodbl= <i>setting</i>	AUTODBL(<i>setting</i>)	Provides an automatic means of converting single-precision floating-point calculations to double-precision and of converting double-precision calculations to extended-precision. Use one of the following settings: none, dbl, dbl4, dbl8, dblpad, dblpad4, or dblpad8. Default: -qautodbl=none	106
-qbigdata -qnobigdata		In 32-bit mode, use this option for programs that exceed 16MB of initialized data (a gcc limitation) and call routines in shared libraries (like open, close, printf and so on). Default: -qnobigdata	166
-qcclines -qnocclines	CCLINES NOCCLINES	Determines whether the compiler recognizes conditional compilation lines. Default: -qcclines if you have specified the -qsmp=omp option; otherwise, -qnocclines.	111
-qctyp _{lss} [=(no)arg] -qnoctyp _{lss}	CTYPLSS [[NO]ARG] NOCTYPLSS	Specifies whether character constant expressions are allowed wherever typeless constants may be used. This language extension might be needed when you are porting programs from other platforms. Suboption <i>arg</i> specifies that Hollerith constants used as actual arguments will be treated as integer actual arguments. Default: -qnoctyp _{lss}	116

Table 7. Options for Compatibility (continued)

Command-Line Option	@PROCESS Directive	Description	See Page
-qddim -qnoddim	DDIM NODDIM	Specifies that the bounds of pointer arrays are re-evaluated each time the arrays are referenced and removes some restrictions on the bounds expressions for pointer arrays. Default: -qnoddim	119
-qdpc -qdpc=e -qnodpc	DPC DPC(E) NODPC	Increases the precision of real constants, for maximum accuracy when assigning real constants to DOUBLE PRECISION variables. This language extension might be needed when you are porting programs from other platforms. Default: -qnodpc	124
-qescape -qnoescape	ESCAPE NOESCAPE	Specifies how the backslash is treated in character strings, Hollerith constants, H edit descriptors, and character string edit descriptors. It can be treated as an escape character or as a backslash character. This language extension might be needed when you are porting programs from other platforms. Default: -qescape	126

Table 7. Options for Compatibility (continued)

Command-Line Option	@PROCESS Directive	Description	See Page
-qextern= <i>names</i>		Allows user-written procedures to be called instead of XL Fortran intrinsics. <i>names</i> is a list of procedure names separated by colons. The procedure names are treated as if they appear in an EXTERNAL statement in each compilation unit being compiled. If any of your procedure names conflict with XL Fortran intrinsic procedures, use this option to call the procedures in the source code instead of the intrinsic ones. Default: Names of intrinsic procedures override user-written procedure names when they are the same.	128
-qextname[= <i>name:name...</i>] -qnoextname	EXTNAME[(<i>name:name...</i>)] NOEXTNAME	Adds an underscore to the names of all global entities, which helps in porting programs from systems where this is a convention for mixed-language programs. Default: -qnoextname	129
-qinit=f90ptr	INIT(f90ptr)	Makes the initial association status of pointers disassociated. Default: The default association status of pointers is undefined.	143
-qintlog -qnointlog	INTLOG NOINTLOG	Specifies that you can mix integer and logical values in expressions and statements. Default: -qnointlog	146
-qintsize= <i>bytes</i>	INTSIZE(<i>bytes</i>)	Sets the size of default INTEGER and LOGICAL values. Default: -qintsize=4	147

Table 7. Options for Compatibility (continued)

Command-Line Option	@PROCESS Directive	Description	See Page
-qlog4 -qnolog4	LOG4 NOLOG4	Specifies whether the result of a logical operation with logical operands is a LOGICAL(4) or is a LOGICAL with the maximum length of the operands. Default: -qnolog4	162
-qminimaltoc -qnominimaltoc		Ensures that the compiler creates only one TOC entry for each compilation unit. By default, the compiler will allocate at least one TOC entry for each unique non-automatic variable reference in your program. Only 8192 TOC entries are available, and duplicate entries are not discarded. This can cause errors when linking large programs in 64-bit mode if your program exceeds 8192 TOC entries. Default: -qnominimaltoc	166
-qmodule=mangle81		Specifies that the compiler should use the XL Fortran Version 8.1 naming convention for non-intrinsic module files. Default: The compiler uses the current naming convention for non-intrinsic module names. This convention is not compatible with that used by previous versions of the compiler.	169
-qnullterm -qnonnullterm	NULLTERM NONULLTERM	Appends a null character to each character constant expression that is passed as a dummy argument, to make it more convenient to pass strings to C functions. Default: -qnonnullterm	171

Table 7. Options for Compatibility (continued)

Command-Line Option	@PROCESS Directive	Description	See Page
-1 -qonetrip -qnoonetrip	ONETRIP NOONETRIP	Executes each DO loop in the compiled program at least once if its DO statement is executed, even if the iteration count is 0. Default: -qnoonetrip	75
-qport [= <i>suboptions</i>] -qnoport	PORT [= <i>suboptions</i>] NOPORT	Increases flexibility when porting programs to XL Fortran by providing a number of options to accommodate other Fortran language extensions. Default: -qnoport	182
-qposition= {appendold appendunknown}	POSITION({APPENDOLD APPENDUNKNOWN})	Positions the file pointer at the end of the file when data is written after an OPEN statement with no POSITION= specifier and the corresponding STATUS= value (OLD or UNKNOWN) is specified. Default: Depends on the I/O specifiers in the OPEN statement and on the compiler invocation command: -qposition=appendold for the xl f , xl f_r , and f77/fort77 commands and the defined Fortran 90 and Fortran 95 behaviors for the xl f90 , xl f90_r , xl f95 , xl f95_r , f90 , and f95 commands.	184
-qqcount -qnoqcount	QCOUNT NOQCOUNT	Accepts the Q character-count edit descriptor (Q) as well as the extended-precision Q edit descriptor (Qw.d). With -qnoqcount , all Q edit descriptors are interpreted as the extended-precision Q edit descriptor. Default: -qnoqcount	186

Table 7. Options for Compatibility (continued)

Command-Line Option	@PROCESS Directive	Description	See Page
-qrealsize= <i>bytes</i>	REALSIZE(<i>bytes</i>)	Sets the default size of REAL , DOUBLE PRECISION , COMPLEX , and DOUBLE COMPLEX values. Default: -qrealsize=4	187
-qsave[={all defaultinit}] -qnosave	SAVE{(ALL DEFAULTINIT)} NOSAVE	Specifies the default storage class for local variables. -qsave , -qsave=all , or -qsave=defaultinit sets the default storage class to STATIC , while -qnosave sets it to AUTOMATIC . Default: -qnosave -qsave is turned on by default for xl , xl_r , f77 , or fort77 to duplicate the behavior of FORTRAN77 commands.	193
-qsclk=[centi micro]		Specifies that when returning a value using the SYSTEM_CLOCK intrinsic procedure, the compiler will use centisecond resolution. You can specify a microsecond resolution by using -qsclk=micro . Default: -qsclk=centi	195
-qswapomp -qnoswapomp	SWAPOMP NOSWAPOMP	Specifies that the compiler should recognize and substitute OpenMP routines in XL Fortran programs. Default: -qswapomp	212
-u -qundef -qnoundef	UNDEF NOUNDEF	Specifies whether implicit typing of variable names is permitted. -u and -qundef have the same effect as the IMPLICIT NONE statement that appears in each scope that allows implicit statements. Default: -qnoundef	236

Table 7. Options for Compatibility (continued)

Command-Line Option	@PROCESS Directive	Description	See Page
-qwarn64 -qnowarn64		Detects the truncation of an 8-byte integer pointer to 4 bytes. Identifies, through informational messages, statements that might cause problems during the 32-bit to 64-bit migration. Default: -qnowarn64	
-qxflag=oldtab	XFLAG(OLDTAB)	Interprets a tab in columns 1 to 5 as a single character (for fixed source form programs). Default: Tab is interpreted as one or more characters.	224
-qxlf77=settings	XLF77(settings)	Provides compatibility with FORTRAN 77 aspects of language semantics and I/O data format that have changed. Most of these changes are required by the Fortran 90 standard. Default: Default suboptions are blankpad, nogedit77, nointarg, nointxor, leadzero, nooldboz, nopersistent, and nosofteof for the xlf90, xlf90_r, xlf95, xlf95_r, f90, and f95 commands and are the exact opposite for the xlf, xlf_r, and f77/fort77 commands.	225
-qxlf90= {[no]signedzero [no]autodealloc}	XLF90({[no]signedzero [no]autodealloc})	Determines whether the compiler provides the Fortran 90 or the Fortran 95 level of support for certain aspects of the language. Default: The default suboptions are signedzero and autodealloc for the xlf95, xlf95_r, and f95 invocation commands. For all other invocation commands, the default suboptions are nosignedzero and noautodealloc.	227

Options for Floating-Point Processing

To take maximum advantage of the system floating-point performance and precision, you may need to specify details of how the compiler and XLF-compiled programs perform floating-point calculations.

Related Information: See “-qflttrap Option” on page 135 and “Duplicating the Floating-Point Results of Other Systems” on page 257.

Table 8. Options for Floating-Point Processing

Command-Line Option	@PROCESS Directive	Description	See Page
-qfloat= <i>options</i>	FLOAT(<i>options</i>)	Determines how the compiler generates or optimizes code to handle particular types of floating-point calculations. Default: Default suboptions are nocomplexgcc , nofltint , fold , nohsflt , maf , nonans , norm , norsqrt , and nostrictnmaf ; some of these settings are different with -O3 optimization turned on or with -qarch=ppc .	133
-qieee={ Near Minus Plus Zero}	IEEE({Near Minus Plus Zero})	Specifies the rounding mode for the compiler to use when evaluating constant floating-point expressions at compile time. Default: -qieee=near	142
-y{n m p z}			

Options That Control Linking

The following options control the way the **ld** command processes object files during compilation. Some of these options are passed on to **ld** and are not processed by the compiler at all.

You can actually include **ld** options on the compiler command line, because the compiler passes unrecognized options on to the linker.

In the table, an * indicates that the option is processed by the **ld** command, rather than the XL Fortran compiler; you can find more information about these options in the man pages for the **ld** command.

Table 9. Options That Control Linking

Command-Line Option	@PROCESS Directive	Description	See Page
-c		Produces an object file instead of an executable file. Default: Compile and link-edit, producing an executable file.	78
-L <i>dir</i> *		Looks in the specified directory for libraries specified by the -l option. Default: /usr/lib	85
-l <i>key</i> *		Searches the specified library file, where <i>key</i> selects the file libkey.so or libkey.a . Default: Libraries listed in xlf.cfg .	86
-qp -qnopic		Generates Position Independent Code (PIC) that can be used in shared libraries. Default: -qnopic in 32-bit mode; -qp=small in 64-bit mode.	181

Options That Control Other Compiler Operations

These options can help to do the following:

- Control internal size limits for the compiler
- Determine names and options for commands that are executed during compilation
- Determine the bit mode and instruction set for the target architecture

Table 10. Options That Control the Compiler Internal Operation

Command-Line Option	@PROCESS Directive	Description	See Page
-B <i>prefix</i>		Determines a substitute path name for executable files used during compilation, such as the compiler or linker. It can be used in combination with the -t option, which determines which of these components are affected by -B . Default: Paths for these components are defined in the configuration file, the <code>\$PATH</code> environment variable, or both.	76
-F <i>config_file</i> -F <i>config_file: stanza</i> -F: <i>stanza</i>		Specifies an alternative configuration file, the stanza to use within the configuration file, or both. Default: The configuration file is <code>/etc/opt/ibmcmp/xlf/9.1/xlf.cfg</code> , and the stanza depends on the name of the command that executes the compiler.	81
-q32		Sets the bit mode and instruction set for a 32-bit target architecture.	244
-q64		Sets the bit mode and instruction set for a 64-bit target architecture.	245
-t <i>components</i>		Applies the prefix specified by the -B option to the designated components. <i>components</i> can be one or more of F, c, d, I, a, h, b, z, or l, with no separators, corresponding to the C preprocessor, the compiler, the -S disassembler, the interprocedural analysis (IPA) tool, the assembler, the loop optimizer, the code generator, the binder, and the linker, respectively. Default: -B prefix, if any, applies to all components.	234
-W <i>component,options</i>		Passes the listed options to a component that is executed during compilation. <i>component</i> is F, c, d, I, a, z, or l, corresponding to the C preprocessor, the compiler, the -S disassembler, the interprocedural analysis (IPA) tool, the assembler, the binder, and the linker, respectively. Default: The options passed to these programs are as follows: <ul style="list-style-type: none"> • Those listed in the configuration file • Any unrecognized options on the command line (passed to the linker) 	239

Options That Are Obsolete or Not Recommended

The following options are obsolete for either or both of the following reasons:

- It has been replaced by an alternative that is considered to be better. Usually this happens when a limited or special-purpose option is replaced by one with a more general purpose and additional features.
- We expect that few or no customers use the feature and that it can be removed from the product in the future with minimal impact to current users.

Notes:

1. If you do use any of these options in existing makefiles or compilation scripts, you should migrate to the new alternatives as soon as you can to avoid any potential problems in the future.
2. The **append** suboption of **-qposition** has been replaced by **appendunknown**.

Table 11. Options That Are Obsolete or Not Recommended

Command-Line Option	@PROCESS Directive	Description	See Page
-qcharlen= <i>length</i>	CHARLEN <i>(length)</i>	Obsolete. It is still accepted, but it has no effect. The maximum length for character constants and subobjects of constants is 32 767 bytes (32 KB). The maximum length for character variables is 268 435 456 bytes (256 MB) in 32-bit mode. The maximum length for character variables is 2**40 bytes in 64-bit mode. These limits are always in effect and are intended to be high enough to avoid portability problems with programs that contain long strings.	
-qrecur -qnorecur	RECUR NORECUR	Not recommended. Specifies whether external subprograms may be called recursively. For new programs, use the RECURSIVE keyword, which provides a standard-conforming way of using recursive procedures. If you specify the -qrecur option, the compiler must assume that any procedure could be recursive. Code generation for recursive procedures may be less efficient. Using the RECURSIVE keyword allows you to specify exactly which procedures are recursive.	189

Detailed Descriptions of the XL Fortran Compiler Options

The following alphabetical list of options provides all the information you should need to use each option effectively.

How to read the syntax information:

- Syntax is shown first in command-line form, and then in **@PROCESS** form if applicable.
- Defaults for each option are underlined and in boldface type.
- Individual required arguments are shown with no special notation.
- When you must make a choice between a set of alternatives, they are enclosed by { and } symbols.
- Optional arguments are enclosed by [and] symbols.
- When you can select from a group of choices, they are separated by | characters.
- Arguments that you can repeat are followed by ellipses (...).

-# Option

Syntax

`-#`

Generates information on the progress of the compilation without actually running the individual components.

Rules

At the points where the compiler executes commands to perform different compilation steps, this option displays a simulation of the system calls it would do and the system argument lists it would pass, but it does not actually perform these actions.

Examining the output of this option can help you quickly and safely determine the following information for a particular compilation:

- What files are involved
- What options are in effect for each step

It avoids the overhead of compiling the source code and avoids overwriting any existing files, such as `.lst` files. (For those who are familiar with the `make` command, it is similar to `make -n`.)

Note that if you specify this option with `-qipa`, the compiler does not display linker information subsequent to the IPA link step. This is because the compiler does not actually call IPA.

Related Information

The “-v Option” on page 237 and “-V Option” on page 238 produce the same output but also performs the compilation.

-1 Option

Syntax

-1
ONETRIP | NOONETRIP

Executes each **DO** loop in the compiled program at least once if its **DO** statement is executed, even if the iteration count is 0. This option provides compatibility with FORTRAN 66. The default is to follow the behavior of later Fortran standards, where **DO** loops are not performed if the iteration count is 0.

Restrictions

It has no effect on **FORALL** statements, **FORALL** constructs, or array constructor implied-**DO** loops.

Related Information

-qonetrp is the long form of -1.

-B Option

Syntax

-Bprefix

Determines a substitute path name for executable files used during compilation, such as the compiler or linker. It can be used in combination with the **-t** option, which determines which of these components are affected by **-B**.

Arguments

prefix is the name of a directory where the alternative executable files reside. It must end in a / (slash).

Rules

To form the complete path name for each component, the driver program adds *prefix* to the standard program names. You can restrict the components that are affected by this option by also including one or more **-tmnemonic** options.

You can also specify default path names for these commands in the configuration file.

This option allows you to keep multiple levels of some or all of the XL Fortran components or to try out an upgraded component before installing it permanently. When keeping multiple levels of XL Fortran available, you might want to put the appropriate **-B** and **-t** options into a configuration-file stanza and to use the **-F** option to select the stanza to use.

Examples

In this example, an earlier level of the XL Fortran components is installed in the directory **/opt/ibmcmp/xlf/8.1/exe**. To test the upgraded product before making it available to everyone, the system administrator restores the latest install image under the directory **/home/jim** and then tries it out with commands similar to:

```
/home/jim/xlf/9.1/bin/xlf95 -tchIbdz -B/home/jim/xlf/9.1/exe/ test_suite.f
```

Once the upgrade meets the acceptance criteria, the system administrator installs it over the old level in **/opt/ibmcmp/xlf/9.1**.

Related Information

See “**-t** Option” on page 234, “**-F** Option” on page 81, “Customizing the Configuration File” on page 14, and “Running Two Levels of XL Fortran” on page 20.

-C Option

Syntax

-C
CHECK | NOCHECK

Checks each reference to an array element, array section, or character substring for correctness.

Rules

At compile time, if the compiler can determine that a reference goes out of bounds, the severity of the error reported is increased to **S** (severe) when this option is specified.

At run time, if a reference goes out of bounds, the program generates a **SIGTRAP** signal. By default, this signal ends the program and produces a core dump. This is expected behaviour and does not indicate there is a defect in the compiler product.

Because the run-time checking can slow execution, you should decide which is the more important factor for each program: the performance impact or the possibility of incorrect results if an error goes undetected. You might decide to use this option only while testing and debugging a program (if performance is more important) or also for compiling the production version (if safety is more important).

Related Information

The **-C** option prevents some of the optimizations that the “-qhot Option” on page 140 performs. You may want to remove the **-C** option after debugging of your code is complete and to add the **-qhot** option to achieve a more thorough optimization.

The valid bounds for character substring expressions differ depending on the setting of the **-qzerosize** option. See “-qzerosize Option” on page 232.

“-qsigtrap Option” on page 197 and “Installing an Exception Handler” on page 258 describe how to detect and recover from **SIGTRAP** signals without ending the program.

-qcheck is the long form of **-C**.

-c Option

Syntax

`-c`

Prevents the completed object file from being sent to the **ld** command for link-editing. With this option, the output is a **.o** file for each source file.

Using the **-o** option in combination with **-c** selects a different name for the **.o** file. In this case, you can only compile one source file at a time.

Related Information

See “**-o Option**” on page 90.

-D Option

Syntax

`-D`
`DLINES` | `NODLINES`

Specifies whether the compiler compiles fixed source form lines with a **D** in column 1 or treats them as comments.

If you specify `-D`, the fixed source form lines that have a **D** in column 1 are compiled. The default action is to treat these lines as comment lines. They are typically used for sections of debugging code that need to be turned on and off.

Related Information

`-qdlines` is the long form of `-D`.

-d Option

Syntax

-d

Causes preprocessed source files that are produced by **cpp** to be kept rather than to be deleted.

Rules

The files that this option produces have names of the form *Ffilename.f*, derived from the names of the original source files.

Related Information

See “Passing Fortran Files through the C Preprocessor” on page 29.

-F Option

Syntax

```
-Fconfig_file | -Fconfig_file:stanza | -F:stanza
```

Specifies an alternative configuration file, which stanza to use within the configuration file, or both.

The configuration file specifies different kinds of defaults, such as options for particular compilation steps and the locations of various files that the compiler requires. A default configuration file (*/etc/opt/ibmcomp/xlf/9.1/xlf.cfg*) is supplied at installation time. The default stanza depends on the name of the command used to invoke the compiler (*xlf90*, *xlf90_r*, *xlf95*, *xlf95_r*, *xlf*, *xlf_r*, *f77*, or *fort77*).

A simple way to customize the way the compiler works, as an alternative to writing complicated compilation scripts, is to add new stanzas to */etc/opt/ibmcomp/xlf/9.1/xlf.cfg*, giving each stanza a different name and a different set of default compiler options. You may find the single, centralized file easier to maintain than many scattered compilation scripts and makefiles.

By running the compiler with an appropriate **-F** option, you can select the set of options that you want. You might have one set for full optimization, another set for full error checking, and so on.

Restrictions

Because the default configuration file is replaced each time a new compiler release is installed, make sure to save a copy of any new stanzas or compiler options that you add.

Examples

```
# Use stanza debug in default xlf.cfg.
xlf95 -F:debug t.f

# Use stanza xlf95 in /home/fred/xlf.cfg.
xlf95 -F/home/fred/xlf.cfg t.f

# Use stanza myxlf in /home/fred/xlf.cfg.
xlf95 -F/home/fred/xlf.cfg:myxlf t.f
```

Related Information

“Customizing the Configuration File” on page 14 explains the contents of a configuration file and tells how to select different stanzas in the file without using the **-F** option.

-g Option

Syntax

`-g`
`DBG | NDBG`

Generates debug information for use by a symbolic debugger.

Related Information

See “Debugging a Fortran 90 or Fortran 95 Program” on page 331 and “Symbolic Debugger Support” on page 9.

`-qdbg` is the long form of `-g`.

-I Option

Syntax

*-I**dir*

Adds a directory to the search path for include files and **.mod** files. If XL Fortran calls **cpp**, this option adds a directory to the search path for **#include** files. Before checking the default directories for include and **.mod** files, the compiler checks each directory in the search path. For include files, this path is only used if the file name in an **INCLUDE** line is not provided with an absolute path. For **#include** files, refer to the **cpp** documentation for the details of the **-I** option.

Arguments

dir must be a valid path name (for example, **/home/dir**, **/tmp**, or **./subdir**).

Rules

The compiler appends a **/** to the *dir* and then concatenates that with the file name before making a search. If you specify more than one **-I** option on the command line, files are searched in the order of the *dir* names as they appear on the command line.

The following directories are searched, in this order, after any paths that are specified by **-I** options:

1. The current directory (from which the compiler is executed)
2. The directory where the source file is (if different from 1)
3. **/usr/include**.

Also, the compiler will search **/opt/ibmcmp/xlf/9.1/include** where include and **.mod** files shipped with the compiler are located.

Related Information

The “-qmoddir Option” on page 168 puts **.mod** files in a specific directory when you compile a file that contains modules.

-k Option

Syntax

-k
FREE(F90)

Specifies that the program is in free source form.

Applicable Product Levels

The meaning of this option has changed from XL Fortran Version 2. To get the old behavior of **-k**, use the option **-qfree=ibm** instead.

Related Information

See “-qfree Option” on page 137 and *Free Source Form* in the *XL Fortran Language Reference*.

This option is the short form of **-qfree=f90**.

-L Option

Syntax

-Ldir

Looks in the specified directory for libraries that are specified by the **-l** option. If you use libraries other than the default ones in **/opt/ibmcomp/xlf/9.1/lib** or **/opt/ibmcomp/xlf/9.1/lib64**, you can specify one or more **-L** options that point to the locations of the other libraries. You can also specify the **LD_LIBRARY_PATH** and **LD_RUN_PATH** environment variables for search paths for libraries.

Rules

This option is passed directly to the **ld** command and is not processed by XL Fortran at all.

Related Information

See “Options That Control Linking” on page 70 and “Linking XL Fortran Programs” on page 32.

-I Option

Syntax

`-Ikey`

Searches the specified library file, where *key* selects the library **libkey.so** or **libkey.a**.

Rules

This option is passed directly to the **ld** command and is not processed by XL Fortran at all.

Related Information

See “Options That Control Linking” on page 70 and “Linking XL Fortran Programs” on page 32.

-N Option

Syntax

-NSbytes
SPILLSIZE(bytes)

Specifies the size of internal program storage areas.

Rules

It defines the number of bytes of stack space to reserve in each subprogram, in case there are too many variables to hold in registers and the program needs temporary storage for register contents.

Defaults

By default, each subprogram stack has 512 bytes of spill space reserved.

If you need this option, a compile-time message informs you of the fact.

Related Information

-qspillsize is the long form of *-NS*.

-O Option

Syntax

`-O[level]`
`OPTimize[(level)]` | **NOOPTimize**

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

Arguments

not specified

Almost all optimizations are disabled. This is equivalent to specifying **-O0** or **-qnoopt**.

-O For each release of XL Fortran, **-O** enables the level of optimization that represents the best tradeoff between compilation speed and run-time performance. If you need a specific level of optimization, specify the appropriate numeric value. Currently, **-O** is equivalent to **-O2**.

-O0 Almost all optimizations are disabled. This option is equivalent to **-qnoopt**.

-O1 Reserved for future use. This form does not currently do any optimization and is ignored. In past releases, it was interpreted as a combination of the **-O** and **-1** options, which may have had unintended results.

-O2 Performs a set of optimizations that are intended to offer improved performance without an unreasonable increase in time or storage that is required for compilation.

-O3 Performs additional optimizations that are memory intensive, compile-time intensive, and may change the semantics of the program slightly, unless **-qstrict** is specified. We recommend these optimizations when the desire for run-time speed improvements outweighs the concern for limiting compile-time resources.

This level of optimization also affects the setting of the **-qfloat** option, turning on the **fltint** and **rsqrt** suboptions by default, and sets **-qmaxmem=-1**.

-O4 Aggressively optimizes the source program, trading off additional compile time for potential improvements in the generated code. You can specify the option at compile time or at link time. If you specify it at link time, it will have no effect unless you also specify it at compile time for at least the file that contains the main program.

-O4 implies the following other options:

- **-qhot**
- **-qipa**
- **-O3** (and all the options and settings that it implies)
- **-qarch=auto**
- **-qtune=auto**
- **-qcache=auto**

Note that the **auto** setting of **-qarch**, **-qtune**, and **-qcache** implies that the execution environment will be the same as the compilation environment.

This option follows the "last option wins" conflict resolution rule, so any of the options that are modified by **-O4** can be subsequently changed. Specifying **-O4 -qarch=com** allows aggressive intraprocedural optimization while maintaining code portability.

-O5 Provides all of the functionality of the **-O4** option, but also provides the functionality of the **-qipa=level=2** option.

Note: Combining **-O2** and higher optimizations with **-qsmp=omp** invokes additional optimization algorithms, including interprocedural analysis (IPA). IPA optimizations provide opportunities for the compiler to generate additional **fmadd** instructions.

To obtain the same floating-point accuracy for optimized and non-optimized applications, you must specify the **-qfloat=nomaf** compiler option. In cases where differences in floating-point accuracy still occur after specifying **-qfloat=nomaf**, the **-qstrict** compiler option allows you to exert greater control over changes that optimization can cause in floating-point semantics.

Restrictions

Generally, use the same optimization level for both the compile and link steps. This is important when using either the **-O4** or **-O5** optimization level to get the best run-time performance. For the **-O5** level, all loop transformations (as specified via the **-qhot** option) are done at the link step.

Increasing the level of optimization may or may not result in additional performance improvements, depending on whether the additional analysis detects any further optimization opportunities.

An optimization level of **-O3** or higher can change the behavior of the program and potentially cause exceptions that would not otherwise occur. Use of the **-qstrict** option can eliminate potential changes and exceptions.

If the **-O** option is used in an **@PROCESS** statement, only an optimization level of 0, 2, or 3 is allowed.

Compilations with optimization may require more time and machine resources than other compilations.

The more the compiler optimizes a program, the more difficult it is to debug the program with a symbolic debugger.

Related Information

“**-qstrict** Option” on page 206 shows how to turn off the effects of **-O3** that might change the semantics of a program.

“**-qipa** Option” on page 149, “**-qhot** Option” on page 140, and “**-qpdf** Option” on page 175 turn on additional optimizations that may improve performance for some programs.

“Optimizing XL Fortran Programs” on page 265 discusses technical details of the optimization techniques the compiler uses and some strategies you can use to get maximum performance from your code.

-qOPTimize is the long form of **-O**.

-o Option

Syntax

`-o name`

Specifies a name for the output object, executable, or assembler source file.

To choose the name for an object file, use this option in combination with the `-c` option. For an assembler source file, use it in combination with the `-S` option.

Defaults

The default name for an executable file is **a.out**. The default name for an object or assembler source file is the same as the source file except that it has a `.o` or `.s` extension.

Rules

Except when you specify the `-c` or `-S` option, the `-o` option is passed directly to the `ld` command, instead of being processed by XL Fortran.

Examples

```
xlf95 t.f                # Produces "a.out"  
xlf95 -c t.f            # Produces "t.o"  
xlf95 -o test_program t.f # Produces "test_program"  
xlf95 -S -o t2.s t.f    # Produces "t2.s"
```

-p Option

Syntax

-p[g]

Sets up the object file for profiling.

-p or **-pg** prepares the program for profiling. When you execute the program, it produces a **gmon.out** file with the profiling information. You can then use the **gprof** command to generate a run-time profile.

Rules

For profiling, 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 the program ends normally, it writes the recorded information to the **gmon.out** file.

Examples

```
$ xlf95 -pg needs_tuning.f
$ a.out
$ gprof
.
```

detailed and verbose profiling data

```
.\n.
```

Related Information

For more information on profiling and the **gprof** command, see the man pages for this command.

-Q Option

Syntax

`-Q+names` | `-Q-names` | `-Q` | `-Q!`

Specifies whether Fortran 90 or Fortran 95 procedures are inlined and/or the names of particular procedures that should or should not be inlined. *names* is a list of procedure names that are separated by colons.

Rules

By default, `-Q` only affects internal or module procedures. To turn on inline expansion for calls to procedures in different scopes, you must also use the `-qipa` option.

Arguments

The `-Q` option without any list inlines all appropriate procedures, subject to limits on the number of inlined calls and the amount of code size increase as a result. `+names` specifies the names, separated by colons, of procedures to inline and raises these limits for those procedures. `-names` specifies the names, separated by colons, of procedures not to inline. You can specify more than one of these options to precisely control which procedures are most likely to be inlined.

The `-Q!` option turns off inlining.

A procedure is not inlined by the basic `-Q` option unless it is quite small. In general, this means that it contains no more than several source statements (although the exact cutoff is difficult to determine). A procedure named by `-Q+` can be up to approximately 20 times larger and still be inlined.

Restrictions

You must specify at least an optimization level of `-O2` for inlining to take effect with `-Q`.

If you specify inlining for a procedure, the following `@PROCESS` compiler directives are only effective if they come before the first compilation unit in the file: `ALIAS`, `ALIGN`, `ATTR`, `COMPACT`, `DBG`, `EXTCHK`, `EXTNAME`, `FLOAT`, `FLTTRAP`, `HALT`, `IEEE`, `LIST`, `MAXMEM`, `OBJECT`, `OPTIMIZE`, `PHSINFO`, `SPILLSIZE`, `STRICT`, and `XREF`.

Examples

```
xlf95 -O -Q many_small_subprogs.f # Compiler decides what to inline.
xlf95 -O -Q+bigfunc:hugefunc test.f # Inline even though these are big.
xlf95 -O -Q -Q-only_once pi.f # Inline except for this one procedure.
```

Related Information

See “`-qipa` Option” on page 149 and “Optimizing Subprogram Calls” on page 277.

-q32 Option

Related Information

See “-q32 Option” on page 244.

-q64 Option

Related Information

See “-q64 Option” on page 245.

-qalias Option

Syntax

```
-qalias={ [no]aryovrlp | [no]intptr | [no]pteovrlp | [no]std }...  
ALIAS( { [NO]ARYOVRLP | [NO]INTPTR | [NO]PTEOVRLP | [NO]STD }... )
```

Indicates whether a program contains certain categories of aliasing. The compiler limits the scope of some optimizations when there is a possibility that different names are aliases for the same storage locations. See “Optimizing XL Fortran Programs” on page 265 for information on aliasing strategies you should consider.

Arguments

aryovrlp | **noaryovrlp**

Indicates whether the compilation units contain any array assignments between storage-associated arrays. If not, specify **noaryovrlp** to improve performance.

intptr | **nointptr**

Indicates whether the compilation units contain any integer **POINTER** statements. If so, specify **INTPTR**.

pteovrlp | **nopteovrlp**

Indicates whether any pointee variables may be used to refer to any data objects that are not pointee variables, or whether two pointee variables may be used to refer to the same storage location. If not, specify **NOPTEOVRLP**.

std | **nostd**

Indicates whether the compilation units contain any nonstandard aliasing (which is explained below). If so, specify **nostd**.

Rules

An alias exists when an item in storage can be referred to by more than one name. The Fortran 90 and Fortran 95 standards allow some types of aliasing and disallow some others. The sophisticated optimizations that the XL Fortran compiler performs increase the likelihood of undesirable results when nonstandard aliasing is present, as in the following situations:

- The same data object is passed as an actual argument two or more times in the same subprogram reference. The aliasing is not valid if either of the actual arguments becomes defined, undefined, or redefined.
- A subprogram reference associates a dummy argument with an object that is accessible inside the referenced subprogram. The aliasing is not valid if any part of the object associated with the dummy argument becomes defined, undefined, or redefined other than through a reference to the dummy argument.
- A dummy argument becomes defined, undefined, or redefined inside a called subprogram, and where the dummy argument was not passed as an actual argument to that subprogram.
- Subscripting beyond the bounds of an array within a common block.

Applicable Product Levels

The introduction of the **-qipa** option does not remove the need for **-qalias**.

Examples

If the following subroutine is compiled with `-qalias=nopteovrlp`, the compiler may be able to generate more efficient code. You can compile this subroutine with `-qalias=nopteovrlp`, because the integer pointers, `ptr1` and `ptr2`, point at dynamically allocated memory only.

```
subroutine sub(arg)
  real arg
  pointer(ptr1, pte1)
  pointer(ptr2, pte2)
  real pte1, pte2

  ptr1 = malloc(%val(4))
  ptr2 = malloc(%val(4))
  pte1 = arg*arg
  pte2 = int(sqrt(arg))
  arg = pte1 + pte2
  call free(%val(ptr1))
  call free(%val(ptr2))
end subroutine
```

If most array assignments in a compilation unit involve arrays that do not overlap but a few assignments do involve storage-associated arrays, you can code the overlapping assignments with an extra step so that the **NOARYOVRLP** suboption is still safe to use.

```
@PROCESS ALIAS(NOARYOVRLP)
! The assertion that no array assignments involve overlapping
! arrays allows the assignment to be done without creating a
! temporary array.
program test
  real(8) a(100)
  integer :: j=1, k=50, m=51, n=100

  a(1:50) = 0.0d0
  a(51:100) = 1.0d0

  ! Timing loop to achieve accurate timing results
  do i = 1, 1000000
    a(j:k) = a(m:n)    ! Here is the array assignment
  end do

  print *, a
end program
```

In Fortran, this aliasing is not permitted if **J** or **K** are updated, and, if it is left undetected, it can have unpredictable results.

! We cannot assert that this unit is free
! of array-assignment aliasing because of the assignments below.

```
subroutine sub1
  integer a(10), b(10)
  equivalence (a, b(3))
  a = b          ! a and b overlap.
  a = a(10:1:-1) ! The elements of a are reversed.
end subroutine
```

! When the overlapping assignment is recoded to explicitly use a
! temporary array, the array-assignment aliasing is removed.
! Although ALIAS(NOARYOVRLP) does not speed up this assignment,
! subsequent assignments of non-overlapping arrays in this unit
! are optimized.

```
@PROCESS ALIAS(NOARYOVRLP)
subroutine sub2
  integer a(10), b(10), t(10)
```

```

equivalence (a, b(3))
t = b; a = t
t = a(10:1:-1); a = t
end subroutine

```

When **SUB1** is called, an alias exists between **J** and **K**. **J** and **K** refer to the same item in storage.

```

CALL SUB1(I,I)
...
SUBROUTINE SUB1(J,K)

```

In the following example, the program might store 5 instead of 6 into **J** unless **-qalias=nostd** indicates that an alias might exist.

```

INTEGER BIG(1000)
INTEGER SMALL(10)
COMMON // BIG
EQUIVALENCE(BIG,SMALL)
...
BIG(500) = 5
SMALL (I) = 6   ! Where I has the value 500
J = BIG(500)

```

Restrictions

Because this option inhibits some optimizations of some variables, using it can lower performance.

Programs that contain nonstandard or integer **POINTER** aliasing may produce incorrect results if you do not compile them with the correct **-qalias** settings. The **xlf90**, **xlf90_r**, **xlf95**, **xlf95_r**, **f90**, and **f95** commands assume that a program contains only standard aliasing (**-qalias=aryovrlp:pteovrlp:std:nointptr**), while the **xlf_r**, **xlf**, and **f77/fort77** commands assume that integer **POINTER**s may be present (**-qalias=aryovrlp:pteovrlp:std:intptr**).

-qalign Option

Syntax

```
-qalign={ [no]4k | struct={suboption} | bindc={suboption} }  
ALIGN({ [NO]4K | STRUCT{(suboption)} | BINDC{(suboption)} })
```

Specifies the alignment of data objects in storage, which avoids performance problems with misaligned data. The **[no]4k**, **bindc**, and **struct** options can be specified and are not mutually exclusive. The **[no]4k** option is useful primarily in combination with logical volume I/O and disk striping.

Defaults

The default setting is **-qalign= no4k:struct=natural:bindc=linuxppc**.

Arguments

[no]4K

Specifies whether to align large data objects on page (4 KB) boundaries, for improved performance with data-striped I/O. Objects are affected depending on their representation within the object file. The affected objects are arrays and structures that are 4 KB or larger and are in static or bss storage and also CSECTs (typically **COMMON** blocks) that are 8 KB or larger. A large **COMMON** block, equivalence group containing arrays, or structure is aligned on a page boundary, so the alignment of the arrays depends on their position within the containing object. Inside a structure of non-sequence derived type, the compiler adds padding to align large arrays on page boundaries.

bindc={suboption}

Specifies that the alignment and padding for an XL Fortran derived type with the BIND(C) attribute is compatible with a C struct type that is compiled with the corresponding XL C alignment option. The compatible alignment options include:

XL Fortran Option	Corresponding XL C Option
-qalign=bindc=bit_packed	-qalign=bit_packed
-qalign=bindc=linuxppc	-qalign=linuxppc

struct={suboption}

The struct option specifies how objects or arrays of a derived type declared using a record structure are stored, and whether or not padding is used between components. All program units must be compiled with the same settings of the **-qalign=struct** option. The three suboptions available are:

packed

If the **packed** suboption of the **struct** option is specified, objects of a derived type are stored with no padding between components, other than any padding represented by **%FILL** components. The storage format is the same as would result for a sequence structure whose derived type was declared using a standard derived type declaration.

natural

If the **natural** suboption of the **struct** option is specified, objects of a derived type are stored with sufficient padding that components will be stored on their natural alignment boundaries, unless storage association requires otherwise. The natural alignment

boundaries for objects of a type that appears in the left-hand column of the following table is shown in terms of a multiple of some number of bytes in the corresponding entry in the right-hand column of the table.

Type	Natural Alignment (in multiples of bytes)
INTEGER(1), LOGICAL(1), BYTE, CHARACTER	1
INTEGER(2), LOGICAL(2)	2
INTEGER(4), LOGICAL(4), REAL(4)	4
INTEGER(8), LOGICAL(8), REAL(8), COMPLEX(4)	8
REAL(16), COMPLEX(8), COMPLEX(16)	16
Derived	Maximum alignment of its components

If the **natural** suboption of the **struct** option is specified, arrays of derived type are stored so that each component of each element is stored on its natural alignment boundary, unless storage association requires otherwise.

port

If the **port** suboption of the **struct** option is specified,

- Storage padding is the same as described above for the **natural** suboption, with the exception that the alignment of components of type complex is the same as the alignment of components of type real of the same kind.
- The padding for an object that is immediately followed by a union is inserted at the beginning of the first map component for each map in that union.

Restrictions

The **port** suboption does not affect any arrays or structures with the **AUTOMATIC** attribute or arrays that are allocated dynamically. Because this option may change the layout of non-sequence derived types, when compiling programs that read or write such objects with unformatted files, use the same setting for this option for all source files.

Related Information

You can tell if an array has the **AUTOMATIC** attribute and is thus unaffected by **-qalign=4k** if you look for the keywords **AUTOMATIC** or **CONTROLLED AUTOMATIC** in the listing of the “-qattr Option” on page 105. This listing also shows the offsets of data objects.

-qarch Option

Syntax

`-qarch=architecture`

Controls which instructions the compiler can generate. Changing the default can improve performance but might produce code that can only be run on specific machines.

In general, the **-qarch** option allows you to target a specific architecture for the compilation. For any given **-qarch** setting, the compiler defaults to a specific, matching **-qtune** setting, which can provide additional performance improvements. The resulting code may not run on other architectures, but it will provide the best performance for the selected architecture. To generate code that can run on more than one architecture, specify a **-qarch** suboption, such as **com**, **ppc**, or **ppc64**, that supports a group of architectures; doing this will generate code that runs on all supported architectures, PowerPC, or 64-bit PowerPC architectures, respectively. When a **-qarch** suboption is specified with a group argument, you can specify **-qtune** as either **auto**, or provide a specific architecture in the group. In the case of **-qtune=auto**, the compiler will generate code that runs on all architectures in the group specified by the **-qarch** suboption, but select instruction sequences that have best performance on the architecture of the machine used to compile. Alternatively you can target a specific architecture for tuning performance.

Arguments

Note: For Y-HPC, only **-qarch=auto** and **-qarch=ppc970** are valid choices for *architecture*; **-qarch=ppc970** is the default.

Otherwise, the choices for *architecture* are:

auto Automatically detects the specific architecture of the compiling machine. It assumes that the execution environment will be the same as the compilation environment.

com You can run the executable file that the compiler generated on any hardware platform supported by the compiler, because the file contains only instructions that are common to all machines. This choice is the same as the default, **-qarch=ppc64grsq**.

If you specify the **-q64** and **-qarch=com** options together, the target platform is 64-bit, and the **-qarch** option is silently upgraded to **ppc64grsq**. The instruction set will be restricted to those instructions common to all 64-bit machines. See “Using XL Fortran in a 64-Bit Environment” on page 243 for details.

ppc You can run the executable file on any PowerPC hardware platform, including those that are based on the RS64II, RS64III, POWER3, POWER4, POWER5, PowerPC 970, and future PowerPC chips. If you specify the compiler option **-q64**, the target platform is 64-bit PowerPC, and the compiler silently upgrades the **-qarch** setting to **ppc64grsq**. See “Using XL Fortran in a 64-Bit Environment” on page 243 for details.

ppcgr In 32-bit mode, produces object code that may contain optional graphics instructions for PowerPC hardware platforms.

In 64-bit mode, produces object code containing optional graphics instructions that will run on 64-bit PowerPC platforms, but not on 32-bit-only platforms, and the **-qarch** option will be silently upgraded to **-qarch=ppc64grsq**.

ppc64 You can run the executable file on any 64-bit PowerPC hardware platform. This suboption can be selected when compiling in 32-bit mode, but the resulting object code may include instructions that are not recognized or behave differently when run on PowerPC platforms that do not support 64-bit mode.

ppc64gr You can run the executable file on any 64-bit PowerPC hardware platform that supports the optional graphics instructions.

ppc64grsq You can run the executable file on any 64-bit PowerPC hardware platform that supports the optional graphics and square root instructions. This is the default option.

rs64b You can run the executable file on any RS64II machine.

rs64c You can run the executable file on any RS64III machine.

pwr3 You can run the executable file on any POWER3, POWER4, POWER5, or PowerPC 970 hardware platform. In previous releases, the **pwr3** setting was used to target the POWER3 and POWER4 group of processors. To have your compilation target a more general processor group, use the **ppc64grsq** setting, which includes the POWER3, POWER4, POWER5, or PowerPC 970 group of processors. Because executable files for these platforms may contain instructions that are not available on other PowerPC systems, they may be incompatible with those systems.

pwr4 You can run the executable file on any POWER4, POWER5, or PowerPC 970 hardware platform. Use of **-qarch=pwr4** will result in binaries that will not run on most previous PowerPC implementations.

pwr5 You can run the executable file on any POWER5 hardware platform.

ppc970 You can run the executable file on any PowerPC 970 hardware platform.

Note: The **-qarch** setting determines the allowed choices and defaults for the **-qtune** setting. You can use **-qarch** and **-qtune** to target your program to particular machines.

If you intend your program to run only on a particular architecture, you can use the **-qarch** option to instruct the compiler to generate code specific to that architecture. This allows the compiler to take advantage of machine-specific instructions that can improve performance. The **-qarch** option provides arguments for you to specify certain chip models; for example, you can specify **-qarch=pwr3** to indicate that your program will be executed on POWER3 hardware platforms.

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

You can further enhance the performance of programs intended for specific machines by using other performance-related options like the **-qcache** and **-qhot** options.

Use these guidelines to help you decide whether to use this option:

- If your primary concern is to make a program widely distributable, keep the default (**ppc64grsq**). If your program is likely to be run on all types of processors equally often, do not specify any **-qarch** or **-qtune** options. The default supports only the common subset of instructions of all processors.
- If you want your program to run on more than one architecture, but to be tuned to a particular architecture, use a combination of the **-qarch** and **-qtune** options. Make sure that your **-qarch** setting covers all the processor types you intend your program to run on.
- If the program will only be used on a single machine or can be recompiled before being used on a different machine, specify the applicable **-qarch** setting. Doing so might improve performance and is unlikely to increase compile time. If you specify the **rs64b**, **rs64c**, **pwr3**, **pwr4**, **pwr5**, or **ppc970** suboption, you do not need to specify a separate **-qtune** option.
- If your primary concern is execution performance, you may see some speedup if you specify the appropriate **-qarch** suboption and perhaps also specify the **-qtune** and **-qcache** options. In this case, you may need to produce different versions of the executable file for different machines, which might complicate configuration management. You will need to test the performance gain to see if the additional effort is justified.
- It is usually better to target a specific architecture so your program can take advantage of the targeted machine's characteristics. For example, specifying **-qarch=pwr4** when targeting a POWER4 machine will benefit those programs that are floating-point intensive or have integer multiplies. On PowerPC systems, programs that process mainly unpromoted single-precision variables are more efficient when you specify **-qarch=ppc**. On POWER3 systems, programs that process mainly double-precision variables (or single-precision variables promoted to double by one of the **-qautodbl** options) become more efficient with **-qarch=pwr3**, **-qarch=pwr4** and **-qarch=pwr5**.

Other Considerations

The PowerPC instruction set includes two optional instruction groups that may be implemented by a particular hardware platform, but are not required. These two groups are the graphics instruction group and the sqrt instruction group. Code compiled with specific **-qarch** options (all of which refer to specific PowerPC machines) will run on any equivalent PowerPC machine that has an identical instruction group. The following table illustrates the instruction groups that are included for the various PowerPC machines.

Table 12. Instruction groups for PowerPC platforms

Processor	Graphics group	sqrt group	64-bit
ppc	no	no	no
ppcgr	yes	no	no
ppc64	no	no	yes
ppc64gr	yes	no	yes
ppc64grsq	yes	yes	yes
rs64b	yes	yes	yes
rs64c	yes	yes	yes
pwr3	yes	yes	yes
pwr4	yes	yes	yes
pwr5	yes	yes	yes

Table 12. Instruction groups for PowerPC platforms (continued)

Processor	Graphics group	sqrt group	64-bit
ppc970	yes	yes	yes

Related Information

See “Compiling for Specific Architectures” on page 29, “-qtune Option” on page 216, and “-qcache Option” on page 109.

-qassert Option

Syntax

`-qassert={deps | nodeps | itercnt=n}`

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

Arguments

nodeps Specifies that no loop-carried dependencies exist.

itercnt Specifies a value for unknown loop iteration counts.

Related Information

See “Cost Model for Loop Transformations” on page 273 for background information and instructions for using these assertions. See also the description of the **ASSERT** directive in the *XL Fortran Language Reference*.

-qattr Option

Syntax

`-qattr[=full] | -qnoattr`
`ATTR[(FULL)] | NOATTR`

Specifies whether to produce the attribute component of the attribute and cross-reference section of the listing.

Arguments

If you specify only **-qattr**, only identifiers that are used are reported. If you specify **-qattr=full**, all identifiers, whether referenced or not, are reported.

If you specify **-qattr** after **-qattr=full**, the full attribute listing is still produced.

You can use the attribute listing to help debug problems caused by incorrectly specified attributes or as a reminder of the attributes of each object while writing new code.

Related Information

See “Options That Control Listings and Messages” on page 61 and “Attribute and Cross-Reference Section” on page 336.

-qautodbl Option

Syntax

`-qautodbl=setting`
`AUTODBL(setting)`

Provides an automatic means of converting single-precision floating-point calculations to double-precision and of converting double-precision calculations to extended-precision.

You might find this option helpful in porting code where storage relationships are significant and different from the XL Fortran defaults. For example, programs that are written for the IBM VS FORTRAN compiler may rely on that compiler's equivalent option.

Arguments

The `-qautodbl` suboptions offer different strategies to preserve storage relationships between objects that are promoted or padded and those that are not.

The settings you can use are as follows:

<u>none</u>	Does not promote or pad any objects that share storage. This setting is the default.
dbl4	Promotes floating-point objects that are single-precision (4 bytes in size) or that are composed of such objects (for example, COMPLEX or array objects): <ul style="list-style-type: none">• REAL(4) is promoted to REAL(8).• COMPLEX(4) is promoted to COMPLEX(8). This suboption requires the libxlfpm4.a library during linking.
dbl8	Promotes floating-point objects that are double-precision (8 bytes in size) or that are composed of such objects: <ul style="list-style-type: none">• REAL(8) is promoted to REAL(16).• COMPLEX(8) is promoted to COMPLEX(16). This suboption requires the libxlfpm8.a library during linking.
dbl	Combines the promotions that dbl4 and dbl8 perform. This suboption requires the libxlfpm4.a and libxlfpm8.a libraries during linking.
dblpad4	Performs the same promotions as dbl4 and pads objects of other types (except CHARACTER) if they could possibly share storage with promoted objects. This suboption requires the libxlfpm4.a and libxlfpad.a libraries during linking.
dblpad8	Performs the same promotions as dbl8 and pads objects of other types (except CHARACTER) if they could possibly share storage with promoted objects. This suboption requires the libxlfpm8.a and libxlfpad.a libraries during linking.
dblpad	Combines the promotions done by dbl4 and dbl8 and pads objects of other types (except CHARACTER) if they could possibly share storage with promoted objects.

This suboption requires the **libxlfpm4.a**, **libxlfpm8.a**, and **libxlfpad.a** libraries during linking.

Rules

If the appropriate **-qautodbl** option is specified during linking, the program is automatically linked with the necessary extra libraries. Otherwise, you must link them in manually.

- When you have both **REAL(4)** and **REAL(8)** calculations in the same program and want to speed up the **REAL(4)** operations without slowing down the **REAL(8)** ones, use **dbl4**. If you need to maintain storage relationships for promoted objects, use **dblpad4**. If you have few or no **REAL(8)** calculations, you could also use **dblpad**.
- If you want maximum precision of all results, you can use **dbl** or **dblpad**. **dbl4**, **dblpad4**, **dbl8**, and **dblpad8** select a subset of real types that have their precision increased.

By using **dbl4** or **dblpad4**, you can increase the size of **REAL(4)** objects without turning **REAL(8)** objects into **REAL(16)**s. **REAL(16)** is less efficient in calculations than **REAL(8)** is.

The **-qautodbl** option handles calls to intrinsics with arguments that are promoted; when necessary, the correct higher-precision intrinsic function is substituted. For example, if single-precision items are being promoted, a call in your program to **SIN** automatically becomes a call to **DSIN**.

Restrictions

- Because character data is not promoted or padded, its relationship with storage-associated items that are promoted or padded may not be maintained.
- If the storage space for a pointee is acquired through the system routine **malloc**, the size specified to **malloc** should take into account the extra space needed to represent the pointee if it is promoted or padded.
- If an intrinsic function cannot be promoted because there is no higher-precision specific name, the original intrinsic function is used, and the compiler displays a warning message.
- You must compile every compilation unit in a program with the same **-qautodbl** setting.

Related Information

For background information on promotion, padding, and storage/value relationships and for some source examples, see “Implementation Details for **-qautodbl** Promotion and Padding” on page 353.

“**-qrealsize** Option” on page 187 describes another option that works like **-qautodbl**, but it only affects items that are of default kind type and does not do any padding. If you specify both the **-qrealsize** and the **-qautodbl** options, only **-qautodbl** takes effect. Also, **-qautodbl** overrides the **-qdpc** option.

-qbigdata Option

Syntax

`-qbigdata` | `-qnobigdata`

In 32-bit mode, use this compiler option for programs that exceed 16MB of initialized data (a gcc limitation) and call routines in shared libraries (like open, close, printf and so on).

-qcache Option

Syntax

```
-qcache=  
{  
  assoc=number |  
  auto |  
  cost=cycles |  
  level=level |  
  line=bytes |  
  size=Kbytes |  
  type={C|c|D|d|I|i}  
}[:...]
```

Specifies the cache configuration for a specific execution machine. The compiler uses this information to tune program performance, especially for loop operations that can be structured (or *blocked*) to process only the amount of data that can fit into the data cache.

If you know exactly what type of system a program is intended to be executed on and that system has its instruction or data cache configured differently from the default case (as governed by the **-qtune** setting), you can specify the exact characteristics of the cache to allow the compiler to compute more precisely the benefits of particular cache-related optimizations.

For the **-qcache** option to have any effect, you must include the **level** and **type** suboptions and specify at least level 2 of **-O**.

- If you know some but not all of the values, specify the ones you do know.
- If a system has more than one level of cache, use a separate **-qcache** option to describe each level. If you have limited time to spend experimenting with this option, it is more important to specify the characteristics of the data cache than of the instruction cache.
- If you are not sure of the exact cache sizes of the target systems, use relatively small estimated values. It is better to have some cache memory that is not used than to have cache misses or page faults from specifying a cache that is larger than the target system has.

Arguments

assoc=number

Specifies the set associativity of the cache:

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. It assumes that the execution environment will be the same as the compilation environment.

cost=cycles

Specifies the performance penalty that results from a cache miss so that the compiler can decide whether to perform an optimization that might result in extra cache misses.

level=level

Specifies which level of cache is affected:

1 Basic cache

- 2 Level-2 cache or the table lookaside buffer (TLB) if the machine has no level-2 cache
- 3 TLB in a machine that does have a level-2 cache

Other levels are possible but are currently undefined. If a system has more than one level of cache, use a separate **-qcache** option to describe each level.

line=bytes

Specifies the line size of the cache.

size=Kbytes

Specifies the total size of this cache.

type={C|c|D|d|I|i}

Specifies the type of cache that the settings apply to, as follows:

- **C** or **c** for a combined data and instruction cache
- **D** or **d** for the data cache
- **I** or **i** for the instruction cache

Restrictions

If you specify the wrong values for the cache configuration or run the program on a machine with a different configuration, the program may not be as fast as possible but will still work correctly. Remember, if you are not sure of the exact values for cache sizes, use a conservative estimate.

Currently, the **-qcache** option only has an effect when you also specify the **-qhot** option.

Examples

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

```
xlf95 -O3 -qhot -qcache=type=c:level=1:size=8:line=64:assoc=2 file.f
```

To tune performance for a system with two levels of data cache, use two **-qcache** options:

```
xlf95 -O3 -qhot -qcache=type=D:level=1:size=256:line=256:assoc=4 \
-qcache=type=D:level=2:size=512:line=256:assoc=2 file.f
```

To tune performance for a system with two types of cache, again use two **-qcache** options:

```
xlf95 -O3 -qhot -qcache=type=D:level=1:size=256:line=256:assoc=4 \
-qcache=type=I:level=1:size=512:line=256:assoc=2 file.f
```

Related Information

See “-qtune Option” on page 216, “-qarch Option” on page 100, and “-qhot Option” on page 140.

-qcclines Option

Syntax

`-qcclines` | `-qnocclines`
`CCLINES` | `NOCCCLINES`

Determines whether the compiler recognizes conditional compilation lines in fixed source form and F90 free source form. IBM free source form is not supported.

Defaults

The default is `-qcclines` if the `-qsmp=omp` option is turned on; otherwise, the default is `-qnocclines`.

Related Information

See *Conditional Compilation* in the *Language Elements* section of the *XL Fortran Language Reference*.

-qcheck Option

Syntax

`-qcheck` | `-qnocheck`
`CHECK` | `NOCHECK`

`-qcheck` is the long form of the “-C Option” on page 77.

-qci Option

Syntax

```
-qci=numbers  
CI(numbers)
```

Specifies the identification numbers (from 1 to 255) of the **INCLUDE** lines to process. If an **INCLUDE** line has a number at the end, the file is only included if you specify that number in a **-qci** option. The set of identification numbers that is recognized is the union of all identification numbers that are specified on all occurrences of the **-qci** option.

This option allows a kind of conditional compilation because you can put code that is only sometimes needed (such as debugging **WRITE** statements, additional error-checking code, or XLF-specific code) into separate files and decide for each compilation whether to process them.

Examples

```
REAL X /1.0/  
INCLUDE 'print_all_variables.f' 1  
X = 2.5  
INCLUDE 'print_all_variables.f' 1  
INCLUDE 'test_value_of_x.f' 2  
END
```

In this example, compiling without the **-qci** option simply declares **X** and assigns it a value. Compiling with **-qci=1** includes two instances of an include file, and compiling with **-qci=1:2** includes both include files.

Restrictions

Because the optional number in **INCLUDE** lines is not a widespread Fortran feature, using it may restrict the portability of a program.

Related Information

See the section on the **INCLUDE** directive in the *XL Fortran Language Reference*.

-qcompact Option

Syntax

`-qcompact` | `-qnocompact`
`COMPACT` | `NOCOMPACT`

Reduces optimizations that increase code size.

By default, some techniques the optimizer uses to improve performance, such as loop unrolling and array vectorization, may also make the program larger. For systems with limited storage, you can use **-qcompact** to reduce the expansion that takes place. If your program has many loop and array language constructs, using the **-qcompact** option will affect your application's overall performance. You may want to restrict using this option to those parts of your program where optimization gains will remain unaffected.

Rules

With **-qcompact** in effect, other optimization options still work; the reductions in code size come from limiting code replication that is done automatically during optimization.

-qcr Option

Syntax

-qcr | -qnocr

Allows you to control how the compiler interprets the CR (carriage return) character. By default, the CR (Hex value X'0d') or LF (Hex value X'0a') character, or the CRLF (Hex value X'0d0a') combination indicates line termination in a source file. This allows you to compile code written using a Mac OS or DOS/Windows editor.

If you specify **-qnocr**, the compiler recognizes only the LF character as a line terminator. You must specify **-qocr** if you use the CR character for a purpose other than line termination.

-qctyp1ss Option

Syntax

-qctyp1ss[(=[no]arg)] | -qnoctyp1ss
CTYPLSS([(NO]ARG)] | NOCTYPLSS

Specifies whether character constant expressions are allowed wherever typeless constants may be used. This language extension might be needed when you are porting programs from other platforms.

Arguments

arg | noarg Suboptions retain the behavior of **-qctyp1ss**. Additionally, **arg** specifies that Hollerith constants used as actual arguments will be treated as integer actual arguments.

Rules

With **-qctyp1ss**, character constant expressions are treated as if they were Hollerith constants and thus can be used in logical and arithmetic expressions.

Restrictions

- If you specify the **-qctyp1ss** option and use a character-constant expression with the **%VAL** argument-list keyword, a distinction is made between Hollerith constants and character constants: character constants are placed in the rightmost byte of the register and padded on the left with zeros, while Hollerith constants are placed in the leftmost byte and padded on the right with blanks. All of the other **%VAL** rules apply.
- The option does not apply to character expressions that involve a constant array or subobject of a constant array at any point.

Examples

Example 1: In the following example, the compiler option **-qctyp1ss** allows the use of a character constant expression.

```
@PROCESS CTYPLSS
  INTEGER I,J
  INTEGER, PARAMETER :: K(1) = (/97/)
  CHARACTER, PARAMETER :: C(1) = (/A'/)

  I = 4HABCD           ! Hollerith constant
  J = 'ABCD'           ! I and J have the same bit representation

! These calls are to routines in other languages.
  CALL SUB(%VAL('A')) ! Equivalent to CALL SUB(97)
  CALL SUB(%VAL(1HA)) ! Equivalent to CALL SUB(1627389952)"

! These statements are not allowed because of the constant-array
! restriction.
!   I = C // C
!   I = C(1)
!   I = CHAR(K(1))
  END
```

Example 2: In the following example, the variable *J* is passed by reference. The suboption **arg** specifies that the Hollerith constant is passed as if it were an integer actual argument.

```
@PROCESS CTYPLSS(ARG)
  INTEGER :: J

  J = 3HIBM
! These calls are to routines in other languages.
  CALL SUB(J)
  CALL SUB(3HIBM) ! The Hollerith constant is passed as if
                  ! it were an integer actual argument
```

Related Information

See *Hollerith Constants* in the *XL Fortran Language Reference* and “Passing Arguments By Reference or By Value” on page 309.

-qdbg Option

Syntax

-qdbg | -qnodbg
DBG | NODBG

-qdbg is the long form of the “-g Option” on page 82.

-qddim Option

Syntax

-qddim | -qnoddim
DDIM | NODDIM

Specifies that the bounds of pointee arrays are re-evaluated each time the arrays are referenced and removes some restrictions on the bounds expressions for pointee arrays.

Rules

By default, a pointee array can only have dimension declarators containing variable names if the array appears in a subprogram, and any variables in the dimension declarators must be dummy arguments, members of a common block, or use- or host-associated. The size of the dimension is evaluated on entry to the subprogram and remains constant during execution of the subprogram.

With the **-qddim** option:

- The bounds of a pointee array are re-evaluated each time the pointee is referenced. This process is called *dynamic dimensioning*. Because the variables in the declarators are evaluated each time the array is referenced, changing the values of the variables changes the size of the pointee array.
- The restriction on the variables that can appear in the array declarators is lifted, so ordinary local variables can be used in these expressions.
- Pointee arrays in the main program can also have variables in their array declarators.

Examples

```
@PROCESS DDIM
INTEGER PTE, N, ARRAY(10)
POINTER (P, PTE(N))
DO I=1, 10
  ARRAY(I)=I
END DO
N = 5
P = LOC(ARRAY(2))
PRINT *, PTE ! Print elements 2 through 6.
N = 7 ! Increase the size.
PRINT *, PTE ! Print elements 2 through 8.
END
```

-qdirective Option

Syntax

```
-qdirective[=directive_list] | -qnodirective[=directive_list]  
DIRECTIVE[(directive_list)] | NODIRECTIVE[(directive_list)]
```

Specifies sequences of characters, known as trigger constants, that identify comment lines as compiler comment directives.

Background Information

A compiler comment directive is a line that is not a Fortran statement but is recognized and acted on by the compiler. To allow you maximum flexibility, any new directives that might be provided with the XL Fortran compiler in the future will be placed inside comment lines. This avoids portability problems if other compilers do not recognize the directives.

Defaults

The compiler recognizes the default trigger constant **IBM***. Specification of **-qsmp** implies **-qdirective=smp\\$\:\$omp:ibmp**, and, by default, the trigger constants **SMP\$**, **\$OMP**, and **IBMP** are also turned on. If you specify **-qsmp=omp**, the compiler ignores all trigger constants that you have specified up to that point and recognizes only the **\$OMP** trigger constant. Specification of **-qthreaded** implies **-qdirective=ibmt**, and, by default, the trigger constant **IBMT** is also turned on.

Arguments

The **-qnodirective** option with no *directive_list* turns off all previously specified directive identifiers; with a *directive_list*, it turns off only the selected identifiers.

-qdirective with no *directive_list* turns on the default trigger constant **IBM*** if it has been turned off by a previous **-qnodirective**.

Notes

- Multiple **-qdirective** and **-qnodirective** options are additive; that is, you can turn directive identifiers on and off again multiple times.
- One or more *directive_lists* can be applied to a particular file or compilation unit; any comment line beginning with one of the strings in the *directive_list* is then considered to be a compiler comment directive.
- The trigger constants are not case-sensitive.
- The characters (,) , ' , " , ; , = , comma, and blank cannot be part of a trigger constant.
- To avoid wildcard expansion in trigger constants that you might use with these options, you can enclose them in single quotation marks on the command line. For example:

```
xlf95 -qdirective='dbg*' -qnodirective='IBM*' directives.f
```
- This option only affects Fortran directives that the XL Fortran compiler provides, not those that any preprocessors provide.

Examples

```
! This program is written in Fortran free source form.
PROGRAM DIRECTV
INTEGER A, B, C, D, E, F
A = 1 ! Begin in free source form.
B = 2
!OLDSTYLE SOURCEFORM(FIXED)
! Switch to fixed source form for this include file.
      INCLUDE 'set_c_and_d.inc'
!IBM* SOURCEFORM(FREE)
! Switch back to free source form.
E = 5
F = 6
END
```

For this example, compile with the option **-qdirective=oldstyle** to ensure that the compiler recognizes the **SOURCEFORM** directive before the **INCLUDE** line. After processing the include-file line, the program reverts back to free source form, after the **SOURCEFORM(FREE)** statement.

Related Information

See the section on the **SOURCEFORM** directive in the *XL Fortran Language Reference*.

As the use of incorrect trigger constants can generate warning messages or error messages or both, you should check the particular directive statement in the *Directives* section of the *XL Fortran Language Reference* for the suitable associated trigger constant.

-qdirectstorage Option

Syntax

`-qdirectstorage` | `-qnodirectstorage`

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

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. For a program to execute correctly on all PowerPC implementations of cache organization, the programmer should assume that separate instruction and data caches exist, and should program to the separate cache model.

Note: Using the `-qdirectstorage` option together with the `CACHE_ZERO` directive may cause your program to fail, or to produce incorrect results..

-qdlines Option

Syntax

`-qdlines` | `-qnodlines`
`DLINES` | `NODLINES`

`-qdlines` is the long form of the “-D Option” on page 79.

-qdp Option

Syntax

```
-qdp[=e] | -qnodpc  
DPC[(E)] | NODPC
```

Increases the precision of real constants, for maximum accuracy when assigning real constants to **DOUBLE PRECISION** variables. This language extension might be needed when you are porting programs from other platforms.

Rules

If you specify **-qdp**, all basic real constants (for example, 1.1) are treated as double-precision constants; the compiler preserves some digits of precision that would otherwise be lost during the assignment to the **DOUBLE PRECISION** variable. If you specify **-qdp=e**, all single-precision constants, including constants with an e exponent, are treated as double-precision constants.

This option does not affect constants with a kind type parameter specified.

Examples

```
@process nodpc  
  subroutine nodpc  
    real x  
    double precision y  
    data x /1.000000000001/ ! The trailing digit is lost  
    data y /1.000000000001/ ! The trailing digit is lost  
  
    print *, x, y, x .eq. y ! So x is considered equal to y  
  end  
  
@process dpc  
  subroutine dpc  
    real x  
    double precision y  
    data x /1.000000000001/ ! The trailing digit is lost  
    data y /1.000000000001/ ! The trailing digit is preserved  
  
    print *, x, y, x .eq. y ! So x and y are considered different  
  end  
  
program testdpc  
  call nodpc  
  call dpc  
end
```

When compiled, this program prints the following:

```
1.000000000    1.000000000000000000    T  
1.000000000    1.0000000000100009      F
```

showing that with **-qdp** the extra precision is preserved.

Related Information

“-qautodbl Option” on page 106 and “-qrealsize Option” on page 187 are more general-purpose options that can also do what **-qdp** does. **-qdp** has no effect if you specify either of these options.

-qenablevmx Option

Syntax

`-qenablevmx` | `-qnoenablevmx`

Instructs the compiler to generate VMX (Vector Multimedia eXtension) instructions. The `-qhot=simd` and `-qarch=ppc970` compiler options must be specified, as well.

Related Information

See the “-qarch Option” on page 100 and “-qhot Option” on page 140.

-qescape Option

Syntax

```
-qescape | -qnoescape  
ESCAPE | NOESCAPE
```

Specifies how the backslash is treated in character strings, Hollerith constants, H edit descriptors, and character string edit descriptors. It can be treated as an escape character or as a backslash character. This language extension might be needed when you are porting programs from other platforms.

Defaults

By default, the backslash is interpreted as an escape character in these contexts. If you specify **-qnoescape**, the backslash is treated as the backslash character.

The default setting is useful for the following:

- Porting code from another Fortran compiler that uses the backslash as an escape character.
- Including “unusual” characters, such as tabs or newlines, in character data. Without this option, the alternative is to encode the ASCII values (or EBCDIC values, on mainframe systems) directly in the program, making it harder to port.

If you are writing or porting code that depends on backslash characters being passed through unchanged, specify **-qnoescape** so that they do not get any special interpretation. You could also write `\\` to mean a single backslash character under the default setting.

Examples

```
$ # Demonstrate how backslashes can affect the output  
$ cat escape.f  
    PRINT *, 'a\bcde\fg'  
    END  
$ xlf95 escape.f  
** _main === End of Compilation 1 ===  
1501-510  Compilation successful for file escape.f.  
$ a.out  
cde  
  g  
$ xlf95 -qnoescape escape.f  
** _main === End of Compilation 1 ===  
1501-510  Compilation successful for file escape.f.  
$ a.out  
  a\bcde\fg
```

In the first compilation, with the default setting of **-qescape**, `\b` is printed as a backspace, and `\f` is printed as a formfeed character. With the **-qnoescape** option specified, the backslashes are printed like any other character.

Related Information

The list of escape sequences that XL Fortran recognizes is shown in Table 20 on page 307.

-qessl Option

Syntax

-qessl | -qnoessl

Allows the use of the Engineering and Scientific Subroutine Library (ESSL) routines in place of Fortran 90 intrinsic procedures.

The ESSL is a collection of subroutines that provides a wide range of mathematical functions for various scientific and engineering applications. The subroutines are tuned for performance on specific architectures. Some of the Fortran 90 intrinsic procedures have similar counterparts in ESSL. Performance is improved when these Fortran 90 intrinsic procedures are linked with ESSL. In this case, you can keep the interface of Fortran 90 intrinsic procedures, and get the added benefit of improved performance using ESSL.

Rules

Use the ESSL Serial Library when linking with **-lessl**. Use the ESSL SMP Library when linking with **-lesslsmpl**.

Either **-lessl** or **-lesslsmpl** must be used whenever code is being compiled with **-qessl**. ESSL V4.1.1 or above is recommended. It supports both 32-bit and 64-bit environments.

Also, since **libessl.so** and **libesslsmpl.so** have a dependency on **libxlf90_r.so**, compile with **xlf_r**, **xlf90_r**, or **xlf95_r**, which use **libxlf90_r.so** as the default to link. You can also specify **-lxlf90_r** on the link command line if you use the linker directly, or other commands to link.

The following MATMUL function calls may use ESSL routines when **-qessl** is enabled:

```
real a(10,10), b(10,10), c(10,10)
c=MATMUL(a,b)
```

Related Information

The ESSL libraries are not shipped with the XL Fortran compiler. For more information on these libraries, see the following URL:
<http://publib.boulder.ibm.com/clresctr/windows/public/esslbooks.html>.

-qextern Option

Syntax

`-qextern=names`

Allows user-written procedures to be called instead of XL Fortran intrinsics. *names* is a list of procedure names separated by colons. The procedure names are treated as if they appear in an **EXTERNAL** statement in each compilation unit being compiled. If any of your procedure names conflict with XL Fortran intrinsic procedures, use this option to call the procedures in the source code instead of the intrinsic ones.

Arguments

Separate the procedure names with colons.

Applicable Product Levels

Because of the many Fortran 90 and Fortran 95 intrinsic functions and subroutines, you might need to use this option even if you did not need it for FORTRAN 77 programs.

Examples

```
subroutine matmul(res, aa, bb, ext)
  implicit none
  integer ext, i, j, k
  real aa(ext, ext), bb(ext, ext), res(ext, ext), temp
  do i = 1, ext
    do j = 1, ext
      temp = 0
      do k = 1, ext
        temp = temp + aa(i, k) * bb(k, j)
      end do
      res(i, j) = temp
    end do
  end do
end subroutine

implicit none
integer i, j, irand
integer, parameter :: ext = 100
real ma(ext, ext), mb(ext, ext), res(ext, ext)

do i = 1, ext
  do j = 1, ext
    ma(i, j) = float(irand())
    mb(i, j) = float(irand())
  end do
end do

call matmul(res, ma, mb, ext)
end
```

Compiling this program with no options fails because the call to **MATMUL** is actually calling the intrinsic subroutine, not the subroutine defined in the program. Compiling with **-qextern=matmul** allows the program to be compiled and run successfully.

-qextname Option

Syntax

`-qextname[=name1[:name2...]]` | `-qnoextname`
`EXTNAME[(name1: name2:...)]` | `NOEXTNAME`

Adds an underscore to the names of all global entities, which helps in porting programs from systems where this is a convention for mixed-language programs. Use `-qextname=name1[:name2...]` to identify a specific global entity (or entities). For a list of named entities, separate each name with a colon.

The name of a main program is not affected.

The `-qextname` option helps to port mixed-language programs to XL Fortran without modifications. Use of this option avoids naming problems that might otherwise be caused by:

- Fortran subroutines, functions, or common blocks that are named **main**, **MAIN**, or have the same name as a system subroutine
- Non-Fortran routines that are referenced from Fortran and contain an underscore at the end of the routine name

Note: XL Fortran Service and Utility Procedures, such as **flush_** and **dtime_**, have these underscores in their names already. By compiling with the `-qextname` option, you can code the names of these procedures without the trailing underscores.

- Non-Fortran routines that call Fortran procedures and use underscores at the end of the Fortran names
- Non-Fortran external or global data objects that contain an underscore at the end of the data name and are shared with a Fortran procedure

Restrictions

You must compile all the source files for a program, including the source files of any required module files, with the same `-qextname` setting.

If you use the **xlfortility** module to ensure that the Service and Utility subprograms are correctly declared, you must change the name to **xlfortility_extname** when compiling with `-qextname`.

If there is more than one Service and Utility subprogram referenced in a compilation unit, using `-qextname` with no names specified and the **xlfortility_extname** module may cause the procedure declaration check not to work accurately.

Examples

```
@PROCESS EXTNAME
  SUBROUTINE STORE_DATA
    CALL FLUSH(10) ! Using EXTNAME, we can drop the final underscore.
  END SUBROUTINE

@PROCESS(EXTNAME(sub1))
program main
  external :: sub1, sub2
  call sub1()      ! An underscore is added.
  call sub2()      ! No underscore is added.
end program
```

Related Information

This option also affects the names that are specified in several other options, so you do not have to include underscores in their names on the command line. The affected options are “-qextern Option” on page 128, “-Q Option” on page 92, and “-qsigtrap Option” on page 197.

-qfixed Option

Syntax

`-qfixed[=right_margin]`
`FIXED[(right_margin)]`

Indicates that the input source program is in fixed source form and optionally specifies the maximum line length.

The source form specified when executing the compiler applies to all of the input files, although you can switch the form for a compilation unit by using a **FREE** or **FIXED @PROCESS** directive or switch the form for the rest of the file by using a **SOURCEFORM** comment directive (even inside a compilation unit).

For source code from some other systems, you may find you need to specify a right margin larger than the default. This option allows a maximum right margin of 132.

Defaults

`-qfixed=72` is the default for the `xl`, `xl_r`, `f77`, and `fort77` commands. `-qfree=f90` is the default for the `f90`, `f95`, `xl`, `xl_r`, `xl95`, and `xl95_r` commands.

Related Information

See “-qfree Option” on page 137.

For the precise specifications of this source form, see *Fixed Source Form* in the *XL Fortran Language Reference*.

-qflag Option

Syntax

`-qflag=dvzlisting_severity:terminal_severity`
`FLAG(listing_severity,terminal_severity)`

You must specify both *listing_severity* and *terminal_severity*.

Limits the diagnostic messages to those of a specified level or higher. Only messages with severity *listing_severity* or higher are written to the listing file. Only messages with severity *terminal_severity* or higher are written to the terminal. **-w** is a short form for **-qflag=e:e**.

Arguments

The severity levels (from lowest to highest) are:

- i** Informational messages. They explain things that you should know, but they usually do not require any action on your part.
- l** Language-level messages, such as those produced under the **-qlanglvl** option. They indicate possible nonportable language constructs.
- w** Warning messages. They indicate error conditions that might require action on your part, but the program is still correct.
- e** Error messages. They indicate error conditions that require action on your part to make the program correct, but the resulting program can probably still be executed.
- s** Severe error messages. They indicate error conditions that require action on your part to make the program correct, and the resulting program will fail if it reaches the location of the error. You must change the **-qhalt** setting to make the compiler produce an object file when it encounters this kind of error.
- u** Unrecoverable error messages. They indicate error conditions that prevent the compiler from continuing. They require action on your part before you can compile your program.
- q** No messages. A severity level that can never be generated by any defined error condition. Specifying it prevents the compiler from displaying messages, even if it encounters unrecoverable errors.

The **-qflag** option overrides any **-qlanglvl** or **-qsaa** options specified.

Defaults

The default for this option is **i:i** so that you do not miss any important informational messages.

Related Information

See “-qlanglvl Option” on page 156 and “Understanding XL Fortran Error Messages” on page 325.

-qfloat Option

Syntax

`-qfloat=options`
`FLOAT(options)`

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

This option replaces several separate options. For any new code, you should use it instead of **-qfold**, **-qmaf**, or related options.

You should be familiar with the information in “XL Fortran Floating-Point Processing” on page 249 and the IEEE standard before attempting to change any **-qfloat** settings.

Defaults

The default setting uses the suboptions **nocomplexgcc**, **nofltint**, **fold**, **nohsflt**, **maf**, **nonans**, **norm**, **normsrt**, and **nostrictmaf**. Some options change this default, as explained below.

The default setting of each suboption remains in effect unless you explicitly change it. For example, if you select **-qfloat=nofold**, the settings for **nohsflt**, or related options are not affected.

Arguments

The available suboptions each have a positive and negative form, such as **fold** and **nofold**, where the negative form is the opposite of the positive.

The suboptions are as follows:

complexgcc | **nocomplexgcc**

Use Linux conventions when passing or returning complex numbers. This option preserves compatibility with gcc-compiled code.

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 by default if **-qfloat=fltint** is not specified or implied by another option, 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.

The Fortran language does not require checking for floating-point values outside the representable range of integers. In order to improve efficiency, the inline sequence used by **-qfloat=fltint** does not perform this check. If passed a value that is out of range, the inline sequence will produce undefined results.

Although this suboption is turned off by default, it is turned on by the **-O3** optimization level unless you also specify **-qstrict**.

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**.

| **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.

| **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. The
| use of this option when compiling other application programs may
| produce unexpected results without warning. Use with discretion.
| See “Technical Details of the **-qfloat=hsflt** Option” on page 352 for
| details.

| **maf** | **nomaf**

| Makes floating-point calculations faster and more accurate by using
| multiply-add instructions where appropriate. The possible disadvantage is
| that results may not be exactly equivalent to those from similar
| calculations that are performed at compile time or on other types of
| computers. Also, negative zero may be produced.

| **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.

| **rrm** | **normm**

| Turns off compiler optimizations that require the rounding mode to be the
| default, round-to-nearest, at run time. Use this option if your program
| changes the rounding mode by any means, such as by calling the **fpsets**
| procedure. Otherwise, the program may compute incorrect results.

| **rsqrt** | **norsqrt**

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

| Although this suboption is turned off by default, specifying **-O3** turns it on
| unless you also specify **-qstrict**.

| **strictnmaf** | **nostrictnmaf**

| Turns off floating-point transformations that are used to introduce negative
| MAF instructions, as these transformations do not preserve the sign of a
| zero value. By default, the compiler enables these types of transformations.

| To ensure strict semantics, specify both **-qstrict** and **-qfloat=strictnmaf**.

-qfltttrap Option

Syntax

`-qfltttrap[=suboptions] | -qnofltttrap`
`FLTTTRAP[(suboptions)] | NOFLTTTRAP`

Determines what types of floating-point exception conditions to detect at run time. The program receives a **SIGFPE** signal when the corresponding exception occurs.

Arguments

ENable	Turn on checking for the specified exceptions in the main program so that the exceptions generate SIGFPE signals. You must specify this suboption if you want to turn on exception trapping without modifying your source code.
IMPrecise	Only check for the specified exceptions on subprogram entry and exit. This suboption improves performance, but it can make the exact spot of the exception difficult to find.
INEXact	Detect and trap on floating-point inexact if exception-checking is enabled. Because inexact results are very common in floating-point calculations, you usually should not need to turn this type of exception on.
INValid	Detect and trap on floating-point invalid operations if exception-checking is enabled.
NANQ	Detect and trap all quiet not-a-number values (NaNQs) and signaling not-a-number values (NaNSs). Trapping code is generated regardless of specifying the enable or imprecise suboption. This suboption detects all NaN values handled by or generated by floating point instructions, including those not created by invalid operations. This option can impact performance.
OVerflow	Detect and trap on floating-point overflow if exception-checking is enabled.
UNDerflow	Detect and trap on floating-point underflow if exception-checking is enabled.
ZERODivide	Detect and trap on floating-point division by zero if exception-checking is enabled.

Defaults

The **-qfltttrap** option without suboptions is equivalent to **-qfltttrap=ov:und:zero:inv:inex**. However, because this default does not include **enable**, it is probably only useful if you already use **fpsets** or similar subroutines in your source. If you specify **-qfltttrap** more than once, both with and without suboptions, the **-qfltttrap** without suboptions is ignored.

Examples

When you compile this program:

```
REAL X, Y, Z
DATA X /5.0/, Y /0.0/
Z = X / Y
PRINT *, Z
END
```

with the command:

```
xlf95 -qfltrap=zerodivide:enable -qsigtrap divide_by_zero.f
```

the program stops when the division is performed.

The **zerodivide** suboption identifies the type of exception to guard against. The **enable** suboption causes a **SIGFPE** signal when the exception occurs. The **-qsigtrap** option produces informative output when the signal stops the program.

Related Information

See “-qsigtrap Option” on page 197.

See “Detecting and Trapping Floating-Point Exceptions” on page 257 for full instructions on how and when to use the **-qfltrap** option, especially if you are just starting to use it.

-qfree Option

Syntax

```
-qfree[={f90|ibm}]  
FREE[({F90|IBM})]
```

Indicates that the source code is in free source form. The **ibm** and **f90** suboptions specify compatibility with the free source form defined for VS FORTRAN and Fortran 90, respectively. Note that the free source form defined for Fortran 90 also applies to Fortran 95.

The source form specified when executing the compiler applies to all of the input files, although you can switch the form for a compilation unit by using a **FREE** or **FIXED @PROCESS** directive or for the rest of the file by using a **SOURCEFORM** comment directive (even inside a compilation unit).

Defaults

-qfree by itself specifies Fortran 90 free source form.

-qfixed=72 is the default for the **xlfc**, **xlfc_r**, **f77**, and **fort77** commands. **-qfree=f90** is the default for the **f90**, **f95**, **xlfc90**, **xlfc90_r**, **xlfc95**, and **xlfc95_r** commands.

Related Information

See “-qfixed Option” on page 131.

-k is equivalent to **-qfree=f90**.

Fortran 90 free source form is explained in *Free Source Form* in the *XL Fortran Language Reference*. It is the format to use for maximum portability across compilers that support Fortran 90 and Fortran 95 features now and in the future.

IBM free source form is equivalent to the free format of the IBM VS FORTRAN compiler, and it is intended to help port programs from the z/OS[®] platform. It is explained in *IBM Free Source Form* in the *XL Fortran Language Reference*.

-qfullpath Option

Syntax

`-qfullpath` | `-qnofullpath`

Records the full, or absolute, path names of source and include files in object files compiled with debugging information (`-g` option).

If you need to move an executable file into a different directory before debugging it or have multiple versions of the source files and want to ensure that the debugger uses the original source files, use the `-qfullpath` option in combination with the `-g` option so that source-level debuggers can locate the correct source files.

Defaults

By default, the compiler records the relative path names of the original source file in each `.o` file. It may also record relative path names for include files.

Restrictions

Although `-qfullpath` works without the `-g` option, you cannot do source-level debugging unless you also specify the `-g` option.

Examples

In this example, the executable file is moved after being created, but the debugger can still locate the original source files:

```
$ xlf95 -g -qfullpath file1.f file2.f file3.f -o debug_version
...
$ mv debug_version $HOME/test_bucket
$ cd $HOME/test_bucket
$ gdb debug_version
```

Related Information

See “`-g` Option” on page 82.

-qhalt Option

Syntax

`-qhalt=severity`
`HALT(severity)`

Stops before producing any object, executable, or assembler source files if the maximum severity of compile-time messages equals or exceeds the specified severity. *severity* is one of **i**, **l**, **w**, **e**, **s**, **u**, or **q**, meaning informational, language, warning, error, severe error, unrecoverable error, or a severity indicating “don’t stop”.

Arguments

The severity levels (from lowest to highest) are:

- i** Informational messages. They explain things that you should know, but they usually do not require any action on your part.
- l** Language-level messages, such as those produced under the **-qlanglvl** option. They indicate possible nonportable language constructs.
- w** Warning messages. They indicate error conditions that might require action on your part, but the program is still correct.
- e** Error messages. They indicate error conditions that require action on your part to make the program correct, but the resulting program can probably still be executed.
- s** Severe error messages. They indicate error conditions that require action on your part to make the program correct, and the resulting program will fail if it reaches the location of the error. You must change the **-qhalt** setting to make the compiler produce an object file when it encounters this kind of error.
- u** Unrecoverable error messages. They indicate error conditions that prevent the compiler from continuing. They require action on your part before you can compile your program.
- q** No messages. A severity level that can never be generated by any defined error condition. Specifying it prevents the compiler from displaying messages, even if it encounters unrecoverable errors.

Defaults

The default is **-qhalt=s**, which prevents the compiler from generating an object file when compilation fails.

Restrictions

The **-qhalt** option can override the **-qobject** option, and **-qnoobject** can override **-qhalt**.

-qhot Option

Syntax

`-qhot[=suboptions] | -qnohot`
`HOT[=suboptions] | NOHOT`

The **-qhot** compiler option is a powerful alternative to hand tuning that provides opportunities to optimize loops and array language. The **-qhot** compiler option will always attempt to optimize loops, regardless of the suboptions you specify.

If you do not specify an optimization level of 2 or higher when using **-O** and **-qhot**, the compiler assumes **-O2**.

For additional information on loop unrolling, see the Directives for Loop Optimization section in the *XL Fortran Language Reference*.

Array Padding: In XL Fortran, array dimensions that are powers of two can lead to a decrease in cache utilization. The **arraypad** suboption allows the compiler to increase the dimensions of arrays where doing so could improve the efficiency of array-processing loops. This can reduce cache misses and page faults that slow your array processing programs.

Specify **-qhot=arraypad** when your source includes large arrays with dimensions that are powers of 2. This can be particularly effective when the first dimension is a power of 2.

The **-C** option turns off some array optimizations.

Vectorization: The **-qhot** compiler option supports the **simd** and **vector** suboptions that can optimize loops in source code for operations on array data, by ensuring that operations run in parallel where applicable. Though both suboptions can provide an increase in performance, each suboption best applies to a particular type of vectorization.

Short Vectorization: -qhot=simd: The **simd** suboption optimizes array data to run mathematical operations in parallel where the target architecture allows such operations. Parallel operations occur in 16-byte vector registers. The compiler divides vectors that exceed the register length into 16-byte units to facilitate optimization. A 16-byte unit can contain one of the following types of data:

- 4 integers
- 8 2-byte units
- 16 1-byte units

Short vectorization does not support double-precision floating point mathematics, and you must specify **-qarch=ppc970**. Users can typically see benefit when applying the **-qhot=simd** optimization to image processing applications.

Long Vectorization: -qhot=vector: The **vector** suboption optimizes array data to run mathematical operations in parallel where applicable. The compiler uses standard registers with no vector size restrictions. Supporting single and double-precision floating-point mathematics, users can typically see benefit when applying **-qhot=vector** to applications with significant mathematical requirements.

Arguments

arraypad

The compiler pads any arrays where there could be an increase in cache

utilization. Not all arrays will necessarily be padded, and the compiler can pad different arrays by different amounts.

arraypad=*n*

The compiler pads each array in the source. The pad amount must be a positive integer value. Each array will be padded by an integral number of elements. The integral value *n* must be multiples of the largest array element size for effective use of **arraypad**. This value is typically 4, 8, or 16.

When you specify the **arraypad** and **arraypad=*n*** options, the compiler does not check for reshaping or equivalences. If padding takes place, your program can produce unexpected results.

simd | nosimd

converts certain operations in a loop that apply to successive elements of an array into a call to a VMX (Vector Multimedia eXtension) instruction. This call calculates several results at one time, which is faster than calculating each result sequentially.

If you specify **-qhot=nosimd**, the compiler performs optimizations on loops and arrays, but avoids replacing certain code with calls to VMX instructions.

You must specify **-qarch=ppc970**.

vector | novector

The compiler converts certain operations in a loop that apply to successive elements of an array into a call to a routine that is in the **libxlopt.a** library. This call calculates several results at one time, which is faster than calculating each result sequentially.

If you specify **-qhot=novector**, the compiler performs optimizations on loops and arrays, but avoids replacing certain code with calls to vector library routines. The **-qhot=vector** option can affect the precision of your program's results so you should specify either **-qhot=novector** or **-qstrict** if the change in precision is unacceptable to you.

Related Information

"Optimizing Loops and Array Language" on page 272 lists the transformations that are performed.

-qieee Option

Syntax

`-qieee={Near | Minus | Plus | Zero}`
`IEEE({Near | Minus | Plus | Zero})`

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

Arguments

The choices are:

Near	Round to nearest representable number.
Minus	Round toward minus infinity.
Plus	Round toward plus infinity.
Zero	Round toward zero.

This option is intended for use in combination with the XL Fortran subroutine `fpsets` or some other method of changing the rounding mode at run time. It sets the rounding mode that is used for compile-time arithmetic (for example, evaluating constant expressions such as `2.0/3.5`). By specifying the same rounding mode for compile-time and run-time operations, you can avoid inconsistencies in floating-point results.

Note: Compile-time arithmetic is most extensive when you also specify the `-O` option.

If you change the rounding mode to other than the default (round-to-nearest) at run time, be sure to also specify `-qfloat=rrm` to turn off optimizations that only apply in the default rounding mode.

Related Information

See “Selecting the Rounding Mode” on page 254, “-O Option” on page 88, and “-qfloat Option” on page 133.

-qinit Option

Syntax

```
-qinit=f90ptr  
INIT(F90PTR)
```

Makes the initial association status of pointers disassociated. Note that this applies to Fortran 95 as well as Fortran 90, and above.

You can use this option to help locate and fix problems that are due to using a pointer before you define it.

Related Information

See *Pointer Association* in the *XL Fortran Language Reference*.

-qinitauto Option

Syntax

`-qinitauto[=hex_value] | -qnoinitauto`

Initializes each byte or word (4 bytes) of storage for automatic variables to a specific value, depending on the length of the *hex_value*. This helps you to locate variables that are referenced before being defined. For example, by using both the **-qinitauto** option to initialize **REAL** variables with a signaling NAN value and the **-qflttrap** option, it is possible to identify references to uninitialized **REAL** variables at run time.

Setting *hex_value* to zero ensures that all automatic variables are cleared before being used. Some programs assume that variables are initialized to zero and do not work when they are not. Other programs may work if they are not optimized but fail when they are optimized. Typically, setting all the variables to all zero bytes prevents such run-time errors. It is better to locate the variables that require zeroing and insert code in your program to do so than to rely on this option to do it for you. Using this option will generally zero more things than necessary and may result in slower programs.

To locate and fix these errors, set the bytes to a value other than zero, which will consistently reproduce incorrect results. This method is especially valuable in cases where adding debugging statements or loading the program into a symbolic debugger makes the error go away.

Setting *hex_value* to **FF** (255) gives **REAL** and **COMPLEX** variables an initial value of “negative not a number”, or -quiet NAN. Any operations on these variables will also result in quiet NAN values, making it clear that an uninitialized variable has been used in a calculation.

This option can help you to debug programs with uninitialized variables in subprograms; for example, you can use it to initialize **REAL** variables with a signaling NAN value. You can initialize 8-byte **REAL** variables to double-precision signaling NAN values by specifying an 8-digit hexadecimal number, that, when repeated, has a double-precision signaling NAN value. For example, you could specify a number such as **7FBFFFFFFF**, that, when stored in a **REAL(4)** variable, has a single-precision signaling NAN value. The value **7FF7FFFF**, when stored in a **REAL(4)** variable, has a single-precision quiet NAN value. If the same number is stored twice in a **REAL(8)** variable (**7FF7FFFF7FF7FFFF**), it has a double-precision signaling NAN value.

Arguments

- The *hex_value* is a 1-digit to 8-digit hexadecimal (0-F) number.
- To initialize each byte of storage to a specific value, specify 1 or 2 digits for the *hex_value*. If you specify only 1 digit, the compiler pads the *hex_value* on the left with a zero.
- To initialize each word of storage to a specific value, specify 3 to 8 digits for the *hex_value*. If you specify more than 2 but fewer than 8 digits, the compiler pads the *hex_value* on the left with zeros.
- In the case of word initialization, if automatic variables are not a multiple of 4 bytes in length, the *hex_value* may be truncated on the left to fit. For example, if you specify 5 digits for the *hex_value* and an automatic variable is only 1 byte long, the compiler truncates the 3 digits on the left-hand side of the *hex_value* and assigns the two right-hand digits to the variable.

- You can specify alphabetic digits as either upper- or lower-case.

Defaults

- By default, the compiler does not initialize automatic storage to any particular value. However, it is possible that a region of storage contains all zeros.
- If you do not specify a *hex_value* suboption for **-qinitauto**, the compiler initializes the value of each byte of automatic storage to zero.

Restrictions

- Equivalenced variables, structure components, and array elements are not initialized individually. Instead, the entire storage sequence is initialized collectively.

Examples

The following example shows how to perform word initialization of automatic variables:

```
subroutine sub()
integer(4), automatic :: i4
character, automatic :: c
real(4), automatic :: r4
real(8), automatic :: r8
end subroutine
```

When you compile the code with the following option, the compiler performs word initialization, as the *hex_value* is longer than 2 digits:

```
-qinitauto=0cf
```

The compiler initializes the variables as follows, padding the *hex_value* with zeros in the cases of the *i4*, *r4*, and *r8* variables and truncating the first hexadecimal digit in the case of the *c* variable:

Variable	Value
i4	000000CF
c	CF
r4	000000CF
r8	000000CF000000CF

Related Information

See “-qflttrap Option” on page 135 and the section on the **AUTOMATIC** directive in the *XL Fortran Language Reference*.

-qintlog Option

Syntax

```
-qintlog | -qnointlog  
INTLOG | NOINTLOG
```

Specifies that you can mix integer and logical data entities in expressions and statements. Logical operators that you specify with integer operands act upon those integers in a bit-wise manner, and integer operators treat the contents of logical operands as integers.

Restrictions

The following operations do not allow the use of logical variables:

- **ASSIGN** statement variables
- Assigned **GOTO** variables
- **DO** loop index variables
- Implied-**DO** loop index variables in **DATA** statements
- Implied-**DO** loop index variables in either input and output or array constructors
- Index variables in **FORALL** constructs

Examples

```
INTEGER I, MASK, LOW_ORDER_BYTE, TWOS_COMPLEMENT  
I = 32767  
MASK = 255  
! Find the low-order byte of an integer.  
LOW_ORDER_BYTE = I .AND. MASK  
! Find the twos complement of an integer.  
TWOS_COMPLEMENT = .NOT. I  
END
```

Related Information

You can also use the intrinsic functions **IAND**, **IOR**, **IEOR**, and **NOT** to perform bitwise logical operations.

-qintsize Option

Syntax

`-qintsize=bytes`
`INTSIZE(bytes)`

Sets the size of default **INTEGER** and **LOGICAL** data entities (that is, those for which no length or kind is specified).

Background Information

The specified size¹ applies to these data entities:

- **INTEGER** and **LOGICAL** specification statements with no length or kind specified.
- **FUNCTION** statements with no length or kind specified.
- Intrinsic functions that accept or return default **INTEGER** or **LOGICAL** arguments or return values unless you specify a length or kind in an **INTRINSIC** statement. Any specified length or kind must agree with the default size of the return value.
- Variables that are implicit integers or logicals.
- Integer and logical literal constants with no kind specified. If the value is too large to be represented by the number of bytes that you specified, the compiler chooses a size that is large enough. The range for 2-byte integers is $-(2^{*15})$ to $2^{*15}-1$, for 4-byte integers is $-(2^{*31})$ to $2^{*31}-1$, and for 8-byte integers is $-(2^{*63})$ to $2^{*63}-1$.
- Typeless constants in integer or logical contexts.

Allowed sizes for *bytes* are:

- 2
- 4 (the default)
- 8

This option is intended to allow you to port programs unchanged from systems that have different default sizes for data. For example, you might need `-qintsize=2` for programs that are written for a 16-bit microprocessor or `-qintsize=8` for programs that are written for a CRAY computer. The default value of 4 for this option is suitable for code that is written specifically for many 32-bit computers. Note that specifying the `-q64` compiler option does not affect the default setting for `-qintsize`.

Restrictions

This option is not intended as a general-purpose method for increasing the sizes of data entities. Its use is limited to maintaining compatibility with code that is written for other systems.

You might need to add **PARAMETER** statements to give explicit lengths to constants that you pass as arguments.

1. In Fortran 90/95 terminology, these values are referred to as *kind type parameters*.

Examples

In the following example, note how variables, literal constants, intrinsics, arithmetic operators, and input/output operations all handle the changed default integer size.

```
@PROCESS INTSIZE(8)
  PROGRAM INTSIZETEST
    INTEGER I
    I = -9223372036854775807    ! I is big enough to hold this constant.
    J = ABS(I)                 ! So is implicit integer J.
    IF (I .NE. J) THEN
      PRINT *, I, '.NE.', J
    END IF
  END
```

The following example only works with the default size for integers:

```
CALL SUB(17)
END

SUBROUTINE SUB(I)
  INTEGER(4) I                ! But INTSIZE may change "17"
                              ! to INTEGER(2) or INTEGER(8).

  ...
END
```

If you change the default value, you must either declare the variable **I** as **INTEGER** instead of **INTEGER(4)** or give a length to the actual argument, as follows:

```
@PROCESS INTSIZE(8)
  INTEGER(4) X
  PARAMETER(X=17)
  CALL SUB(X)                ! Use a parameter with the right length, or
  CALL SUB(17_4)            ! use a constant with the right kind.
END
```

Related Information

See “-qrealsize Option” on page 187 and *Type Parameters and Specifiers* in the *XL Fortran Language Reference*.

-qipa Option

Syntax

`-qipa[=suboptions] | -qnoipa`

Enhances **-O** optimization by doing detailed analysis across procedures (interprocedural analysis or IPA).

You must also specify the **-O**, **-O2**, **-O3**, **-O4**, or **-O5** option when you specify **-qipa**. (Specifying the **-O5** option is equivalent to specifying the **-O4** option plus **-qipa=level=2**.) For additional performance benefits, you can also specify the **-Q** option. The **-qipa** option extends the area that is examined during optimization and inlining from a single procedure to multiple procedures (possibly in different source files) and the linkage between them.

You can fine-tune the optimizations that are performed by specifying suboptions.

To use this option, the necessary steps are:

1. Do preliminary performance analysis and tuning before compiling with the **-qipa** option. This is necessary because interprocedural analysis uses a two-phase mechanism, a compile-time and a link-time phase, which increases link time. (You can use the **noobject** suboption to reduce this overhead.)
2. Specify the **-qipa** option on both the compile and link steps of the entire application or on as much of it as possible. Specify suboptions to indicate what assumptions to make about the parts of the program that are not compiled with **-qipa**. (If your application contains C or C++ code compiled with IBM XL C/C++ compilers, you must compile with the **-qipa** option to allow for additional optimization opportunities at link time.)

During compilation, the compiler stores interprocedural analysis information in the **.o** file. During linking, the **-qipa** option causes a complete reoptimization of the entire application.

Note that if you specify this option with **#**, the compiler does not display linker information subsequent to the IPA link step. This is because the compiler does not actually call IPA.

Arguments

IPA uses the following suboptions during its compile-time phase:

object | **noobject**

Specifies whether to include standard object code in the object files. Specifying the **noobject** suboption can substantially reduce overall compilation 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.

If your program contains object files created with the **noobject** suboption, you must use the **-qipa** option to compile any files containing an entry point (the main program for an executable program or an exported procedure for a library) before linking your program with **-qipa**.

IPA uses the following suboptions during its link-time phase:

exits=*procedure_names*

Specifies a list of procedures, each of which always ends the program. The

compiler can optimize calls to these procedures (for example, by eliminating save/restore sequences), because the calls never return to the program. These procedures must not call any other parts of the program that are compiled with **-qipa**.

inline=*inline-options*

The **-qipa=inline=** command can take a colon-separated list of inline options, as listed below:

inline=auto | **noauto**

Specifies whether to automatically inline procedures.

inline=limit=number

Changes the size limits that the **inline=auto** option uses to determine how much inline expansion to do. This established "limit" is the size below which the calling procedure must remain. *number* is the optimizer's approximation of the number of bytes of code that will be generated. Larger values for this number allow the compiler to inline larger subprograms, more subprogram calls, or both. This argument is implemented only when **inline=auto** is on.

inline=procedure_names

Specifies a list of procedures to try to inline.

inline=threshold=number

Specifies the upper size limit on procedures to be inlined, where *number* is a value as defined under the inline suboption "limit". This argument is implemented only when "inline=auto" is on.

Note: By default, the compiler will try to inline all procedures, not just those that you specified with the **inline=procedure_names** suboption. If you want to turn on inlining for only certain procedures, specify **inline=noauto** after you specify **inline=procedure_names**. (You must specify the suboptions in this order.) For example, to turn off inlining for all procedures other than for **sub1**, specify **-qipa=inline=sub1:inline=noauto**.

isolated=*procedure_names*

Specifies a comma-separated list of procedures that are not compiled with **-qipa**. Procedures that you specify as "isolated" or procedures within their call chains cannot refer directly to any global variable.

level=*level*

Determines the amount of interprocedural analysis and optimization that is performed:

0 Does only minimal interprocedural analysis and optimization.

1 Turns on inlining, limited alias analysis, and limited call-site tailoring.

2 Full interprocedural data flow and alias analysis. Specifying **-O5** is equivalent to specifying **-O4** and **-qipa=level=2**.

The default level is 1.

list=[*filename* | **short** | **long**]

Specifies an output listing file name during the link phase, in the event that an object listing has been requested using either the **-qlist** or the **-qipa=list** compiler option and allows the user to direct the type of output. If you do not specify the *filename* suboption, the default file name is "a.lst".

If you specify **short**, the Object File Map, Source File Map, and Global Symbols Map sections are included. If you specify **long**, the preceding sections appear in addition to the Object Resolution Warnings, Object Reference Map, Inliner Report, and Partition Map sections.

If you specify the **-qipa** and **-qlist** options together, IPA generates an `a.lst` file that overwrites any existing `a.lst` file. If you have a source file named `a.f`, the IPA listing will overwrite the regular compiler listing `a.lst`. You can use the `list=filename` suboption to specify an alternative listing file name.

lowfreq=*procedure_names*

Specifies a list of procedures that are likely to be called infrequently during the course of a typical program run. For example, procedures for initialization and cleanup might only be called once, and debugging procedures might not be called at all in a production-level program. The compiler can make other parts of the program faster by doing less optimization for calls to these procedures.

missing={unknown | **safe** | **isolated** | **pure**}

Specifies the interprocedural behavior of procedures that are not compiled with **-qipa** and are not explicitly named in an **unknown**, **safe**, **isolated**, or **pure** suboption. The default is to assume **unknown**, which greatly restricts the amount of interprocedural optimization for calls to those procedures.

noinline=*procedure_names*

Specifies a list of procedures that are not to be inlined.

partition={**small** | medium | **large**}

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

pdfname=[*filename*]

Specifies the name of the profile data file containing the **PDF** profiling information. If you do not specify a *filename*, the default file name is `__pdf`. The profile is placed in the current working directory or in the directory that the **PDFDIR** environment variable names. This allows the programmer to do simultaneous runs of multiple executables using the same **PDFDIR**. This is especially useful when tuning with **PDF** on dynamic libraries. (See “-qpdf Option” on page 175 for more information on tuning optimizations.)

pure=*procedure_names*

Specifies a list of procedures that are not compiled with **-qipa**. Any procedure that you specified as “pure” must be “isolated” and “safe”. It must not alter the internal state nor have side-effects, which are defined as potentially altering any data object visible to the caller.

safe=*procedure_names*

Specifies a list of procedures that are not compiled with **-qipa**. Any procedure that you specified as “safe” may modify global variables and dummy arguments. No calls to procedures that are compiled with **-qipa** may be made from within a “safe” procedure’s call chain.

stdexits | nostdexits

Specifies that certain predefined routines can be optimized as with the **exits** suboption. The procedures are: **abort**, **exit**, **_exit**, and **_assert**.

| **threads[=N] | nothreads**

| **threads[=N]** runs the number of parallel threads that are available, or as
| specified by *N*. *N* must be a positive integer. **nothreads** does not run any
| parallel threads. This is equivalent to running one serial thread.

| Specifying **-qipa=threads** can reduce IPA optimization time. The threads
| suboption allows the IPA optimizer to run portions of the optimization
| process in parallel threads, which can speed up the compilation process on
| multi-processor systems.

| **unknown=procedure_names**

| Specifies a list of procedures that are not compiled with **-qipa**. Any
| procedure specified as “unknown” may make calls to other parts of the
| program compiled with **-qipa** and modify global variables and dummy
| arguments.

The primary use of **isolated**, **missing**, **pure**, **safe**, and **unknown** is to specify how much optimization can safely be performed on calls to library routines that are not compiled with **-qipa**.

The following compiler options have an effect on the link-time phase of **-qipa**:

| **-qlibansi | -qnolibansi**

| Assumes that all functions with the name of an ANSI C defined library
| function are, in fact, the library functions.

| **-qlibposix | -qnolibposix**

| Assumes that all functions with the name of an IEEE 1003.1-2001 (POSIX)
| defined library function are, in fact, the system functions.

| **-qthreaded**

| Assumes that the compiler will attempt to generate thread-safe code.

Rules

Regular expressions are supported for the following suboptions:

exits
inline
lowfreq
noinline
pure
safe
unknown

Syntax rules for regular expressions are described below.

Table 13. Regular expression syntax

Expression	Description
string	Matches any of the characters specified in string. For example, test will match testimony, latest, intestine.
^string	Matches the pattern specified by string only if it occurs at the beginning of a line.
string\$	Matches the pattern specified by string only if it occurs at the end of a line.
str.ing	Matches any character. For example, t.st will match test, tast, tZst, and t1st.

Table 13. Regular expression syntax (continued)

Expression	Description
string\.\$	The backslash (\) can be used to escape special characters so that you can match for the character. For example, if you want to find those lines ending with a period, the expression <code>.\$</code> would show all lines that had at least one character. Specify <code>\.\$</code> to escape the period (<code>.</code>).
[string]	Matches any of the characters specified in <code>string</code> . For example, <code>t[a-g123]st</code> matches <code>tast</code> and <code>test</code> , but not <code>t-st</code> or <code>tAst</code> .
[^string]	Does not match any of the characters specified in <code>string</code> . For example, <code>t[^a-zA-Z]st</code> matches <code>t1st</code> , <code>t-st</code> , and <code>t,st</code> but not <code>test</code> or <code>tYst</code> .
string*	Matches zero or more occurrences of the pattern specified by <code>string</code> . For example, <code>te*st</code> will match <code>tst</code> , <code>test</code> , and <code>teeeest</code> .
string+	Matches one or more occurrences of the pattern specified by <code>string</code> . For example, <code>t(es)+t</code> matches <code>test</code> , <code>tesest</code> , but not <code>tt</code> .
string?	Matches zero or more occurrences of the pattern specified by <code>string</code> . For example, <code>te?st</code> matches either <code>tst</code> or <code>test</code> .
string{m,n}	Matches between <code>m</code> and <code>n</code> occurrence(s) of the pattern specified by <code>string</code> . For example, <code>a{2}</code> matches <code>aa</code> , <code>b{1,4}</code> matches <code>b</code> , <code>bb</code> , <code>bbb</code> , and <code>bbbb</code> .
string1 string2	Matches the pattern specified by either <code>string1</code> or <code>string2</code> . For example, <code>s o</code> matches both characters <code>s</code> and <code>o</code> .

Since only function names are being considered, the regular expressions are automatically bracketed with the `^` and `$` characters. For example, `-qipa=noinline=^foo$` is equivalent to `-qipa=noinline=foo`. Therefore, `-qipa=noinline=bar` ensures that `bar` is never inlined but `bar1`, `teebar`, and `barrel` may be inlined.

Examples

The following example shows how you might compile a set of files with interprocedural analysis:

```
xlf95 -O -qipa f.f
xlf95 -c -O3 *.f -qipa=noobject
xlf95 -o product *.o -qipa -O
```

The following example shows how you might link these same files with interprocedural analysis, using regular expressions to improve performance. This example assumes that function `user_abort` exits the program, and that routines `user_trace1`, `user_trace2`, and `user_trace3` are rarely called.

```
xlf95 -o product *.o -qipa=exit=user_abort:lowfreq=user_trace[123] -O
```

Related Information

See the “-O Option” on page 88, “-p Option” on page 91, and “-Q Option” on page 92.

-qkeepparm Option

Syntax

`-qkeepparm` | `-qnokeepparm`

Background Information

A procedure usually stores its incoming parameters on the stack at the entry point. When you compile code with optimization, however, the optimizer may remove the stores into the stack if it sees opportunities to do so.

Specifying the **-qkeepparm** compiler option ensures that the parameters are stored on the stack even when optimizing. This may negatively impact execution performance. This option then provides access to the values of incoming parameters to tools, such as debuggers, simply by preserving those values on the stack.

-qlanglvl Option

Syntax

```
-qlanglvl={suboption}  
LANGLVL({suboption})
```

Determines which language standard (or superset, or subset of a standard) to consult for nonconformance. It identifies nonconforming source code and also options that allow such nonconformances.

Rules

The compiler issues a message with severity code **L** if it detects syntax that is not allowed by the language level that you specified.

Arguments

77std	Accepts the language that the ANSI FORTRAN 77 standard specifies and reports anything else as an error.
90std	Accepts the language that the ISO Fortran 90 standard specifies and reports anything else as an error.
90pure	The same as 90std except that it also reports errors for any obsolescent Fortran 90 features used.
95std	Accepts the language that the ISO Fortran 95 standard specifies and reports anything else as an error.
95pure	The same as 95std except that it also reports errors for any obsolescent Fortran 95 features used.
<u>extended</u>	Accepts the full Fortran 95 language standard and all extensions, effectively turning off language-level checking.

Defaults

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

Restrictions

The **-qflag** option can override this option.

Examples

The following example contains source code that conforms to a mixture of Fortran standards:

```
!-----  
! in free source form  
program tt  
  integer :: a(100,100), b(100), i  
  real :: x, y  
  ...  
  goto (10, 20, 30), i  
10 continue  
  pause 'waiting for input'  
  
20 continue  
  y= gamma(x)  
  
30 continue  
  b = maxloc(a, dim=1, mask=a .lt 0)  
  
end program  
!-----
```

The following chart shows examples of how some **-qlanglvl** suboptions affect this sample program:

-qlanglvl Suboption Specified	Result	Reason
95pure	Flags PAUSE statement Flags computed GOTO statement Flags GAMMA intrinsic	Deleted feature in Fortran 95 Obsolescent feature in Fortran 95 Extension to Fortran 95
95std	Flags PAUSE statement Flags GAMMA intrinsic	Deleted feature in Fortran 95 Extension to Fortran 95
extended	No errors flagged	

Related Information

See “-qflag Option” on page 132, “-qhalt Option” on page 139, and “-qsaa Option” on page 192.

The **langlvl** run-time option, which is described in “Setting Run-Time Options” on page 35, helps to locate run-time extensions that cannot be checked for at compile time.

-qlibansi Option

Related Information

See “-qipa Option” on page 149.

-qlibposix Option

Related Information

See “-qipa Option” on page 149.

-qlist Option

Syntax

`-qlist` | `-qnolist`
`LIST` | `NOLIST`

Specifies whether to produce the object section of the listing.

You can use the object listing to help understand the performance characteristics of the generated code and to diagnose execution problems.

If you specify the `-qipa` and `-qlist` options together, IPA generates an `a.lst` file that overwrites any existing `a.lst` file. If you have a source file named `a.f`, the IPA listing will overwrite the regular compiler listing `a.lst`. To avoid this, use the `list=filename` suboption of `-qipa` to generate an alternative listing.

Related Information

See “Options That Control Listings and Messages” on page 61, “Object Section” on page 337, and “-S Option” on page 233.

-qlistopt Option

Syntax

`-qlistopt` | `-qnolistopt`
`LISTOPT` | `NOLISTOPT`

Determines whether to show the setting of every compiler option in the listing file or only selected options. These selected options include those specified on the command line or directives plus some that are always put in the listing.

You can use the option listing during debugging to check whether a problem occurs under a particular combination of compiler options or during performance testing to record the optimization options in effect for a particular compilation.

Rules

Options that are always displayed in the listing are:

- All “on/off” options that are on by default: for example, `-qobject`
- All “on/off” options that are explicitly turned off through the configuration file, command-line options, or `@PROCESS` directives
- All options that take arbitrary numeric arguments (typically sizes)
- All options that have multiple suboptions

Related Information

See “Options That Control Listings and Messages” on page 61 and “Options Section” on page 333.

-qlog4 Option

Syntax

`-qlog4` | `-qnolog4`
`LOG4` | `NOLOG4`

Specifies whether the result of a logical operation with logical operands is a **LOGICAL(4)** or is a **LOGICAL** with the maximum length of the operands.

You can use this option to port code that was originally written for the IBM VS FORTRAN compiler.

Arguments

`-qlog4` makes the result always a **LOGICAL(4)**, while `-qnolog4` makes it depend on the lengths of the operands.

Restrictions

If you use `-qintsize` to change the default size of logicals, `-qlog4` is ignored.

-qmaxmem Option

Syntax

`-qmaxmem=Kbytes`
`MAXMEM(Kbytes)`

Limits the amount of memory that the compiler allocates while performing specific, memory-intensive optimizations to the specified number of kilobytes. A value of -1 allows optimization to take as much memory as it needs without checking for limits.

Defaults

At the **-O2** optimization level, the default **-qmaxmem** setting is 2048 KB. At the **-O3** optimization level, the default setting is unlimited (-1).

Rules

If the specified amount of memory is insufficient for the compiler to compute a particular optimization, the compiler issues a message and reduces the degree of optimization.

This option has no effect except in combination with the **-O** option.

When compiling with **-O2**, you only need to increase the limit if a compile-time message instructs you to do so. When compiling with **-O3**, you might need to establish a limit if compilation stops because the machine runs out of storage; start with a value of 2048 or higher, and decrease it if the compilation continues to require too much storage.

Notes:

1. Reduced optimization does not necessarily mean that the resulting program will be slower. It only means that the compiler cannot finish looking for opportunities to improve performance.
2. Increasing the limit does not necessarily mean that the resulting program will be faster. It only means that the compiler is better able to find opportunities to improve performance if they exist.
3. Setting a large limit has no negative effect when compiling source files for which the compiler does not need to use so much memory during optimization.
4. As an alternative to raising the memory limit, you can sometimes move the most complicated calculations into procedures that are then small enough to be fully analyzed.
5. Not all memory-intensive compilation stages can be limited.
6. Only the optimizations done for **-O2** and **-O3** can be limited; **-O4** and **-O5** optimizations cannot be limited.
7. The **-O4** and **-O5** optimizations may also use a file in the `/tmp` directory. This is not limited by the **-qmaxmem** setting.
8. Some optimizations back off automatically if they would exceed the maximum available address space, but not if they would exceed the paging space available at that time, which depends on machine workload.

Restrictions

Depending on the source file being compiled, the size of subprograms in the source code, the machine configuration, and the workload on the system, setting the limit too high might fill up the paging space. In particular, a value of -1 can fill up the storage of even a well-equipped machine.

Related Information

See “-O Option” on page 88 and “Optimizing XL Fortran Programs” on page 265.

-qmbcs Option

Syntax

-qmbcs | -qnombcs
MBCS | NOMBCS

Indicates to the compiler whether character literal constants, Hollerith constants, H edit descriptors, and character string edit descriptors can contain Multibyte Character Set (MBCS) or Unicode characters.

This option is intended for applications that must deal with data in a multibyte language, such as Japanese.

To process the multibyte data correctly at run time, set the locale (through the **LANG** environment variable or a call to the **libc setlocale** routine) to the same value as during compilation.

Rules

Each byte of a multibyte character is counted as a column.

Restrictions

To read or write Unicode data, set the locale value to **UNIVERSAL** at run time. If you do not set the locale, you might not be able to interchange data with Unicode-enabled applications.

-qminimaltoc Option

Syntax

`-qminimaltoc` | `-qnomimaltoc`

This compiler option can change the generation of the table of contents (TOC), which the compiler creates for every executable file when you compile in 64-bit mode. By default, the compiler will allocate at least one TOC entry for each unique, non-automatic variable reference in your program. Currently, only 8192 TOC entries are available and duplicate entries are not discarded. This can cause errors when linking large programs in 64-bit mode if your program exceeds 8192 TOC entries.

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.

-qmixed Option

Syntax

`-qmixed` | `-qmixed`
`MIXED` | `NOMIXED`

This is the long form of the “-U Option” on page 235.

-qmoddir Option

Syntax

`-qmoddir=directory`

Specifies the location for any module (**.mod**) files that the compiler writes.

Defaults

If you do not specify **-qmoddir**, the **.mod** files are placed in the current directory.

To read the **.mod** files from this directory when compiling files that reference the modules, use the “**-I Option**” on page 83.

Related Information

See “**XL Fortran Output Files**” on page 25.

Modules are a Fortran 90/95 feature and are explained in the Modules section of the *XL Fortran Language Reference*.

-qmodule Option

Syntax

`-qmodule=mangle81`

Specifies that the compiler should use the XL Fortran Version 8.1 naming convention for non-intrinsic module files.

This option allows you to produce modules and their associated object files with the Version 9.1 compiler and link these object files with others compiled with the Version 8.1 compiler, or earlier.

Related Information

Modules are a Fortran 90/95 feature and are explained in the Modules section of the *XL Fortran Language Reference*.

See also “Conventions for XL Fortran External Names” on page 301 and “Avoiding Naming Conflicts during Linking” on page 33.

-qnoprint Option

Syntax

`-qnoprint`

Prevents the compiler from creating the listing file, regardless of the settings of other listing options.

Specifying **-qnoprint** on the command line enables you to put other listing options in a configuration file or on **@PROCESS** directives and still prevent the listing file from being created.

Rules

A listing file is usually created when you specify any of the following options: **-qattr**, **-qlist**, **-qlistopt**, **-qphsinfo**, **-qsource**, **-qreport**, or **-qxref**. **-qnoprint** prevents the listing file from being created by changing its name to **/dev/null**, a device that discards any data that is written to it.

Related Information

See "Options That Control Listings and Messages" on page 61.

-qnullterm Option

Syntax

```
-qnullterm | -qnonullterm  
NULLTERM | NONULLTERM
```

Appends a null character to each character constant expression that is passed as a dummy argument, to make it more convenient to pass strings to C functions.

This option allows you to pass strings to C functions without having to add a null character to each string argument.

Background Information

This option affects arguments that are composed of any of the following objects: basic character constants; concatenations of multiple character constants; named constants of type character; Hollerith constants; binary, octal, or hexadecimal typeless constants when an interface block is available; or any character expression composed entirely of these objects. The result values from the **CHAR** and **ACHAR** intrinsic functions also have a null character added to them if the arguments to the intrinsic function are initialization expressions.

Rules

This option does not change the length of the dummy argument, which is defined by the additional length argument that is passed as part of the XL Fortran calling convention.

Restrictions

This option affects those arguments that are passed with or without the **%REF** built-in function, but it does not affect those that are passed by value. This option does not affect character expressions in input and output statements.

Examples

Here are two calls to the same C function, one with and one without the option:

```
@PROCESS NONULLTERM  
SUBROUTINE CALL_C_1  
  CHARACTER*9, PARAMETER :: HOME = "/home/luc"  
! Call the libc routine mkdir() to create some directories.  
  CALL mkdir ("/home/luc/testfiles\0", %val(448))  
! Call the libc routine unlink() to remove a file in the home directory.  
  CALL unlink (HOME // "/.hushlogin" // CHAR(0))  
END SUBROUTINE  
  
@PROCESS NULLTERM  
SUBROUTINE CALL_C_2  
  CHARACTER*9, PARAMETER :: HOME = "/home/luc"  
! With the option, there is no need to worry about the trailing null  
! for each string argument.  
  CALL mkdir ("/home/luc/testfiles", %val(448))  
  CALL unlink (HOME // "/.hushlogin")  
END SUBROUTINE  
!
```

Related Information

See "Passing Character Types between Languages" on page 306.

-qobject Option

Syntax

-qOBJect | **-qNOOBJect**
OBJect | **NOOBJect**

Specifies whether to produce an object file or to stop immediately after checking the syntax of the source files.

When debugging a large program that takes a long time to compile, you might want to use the **-qnoobject** option. It allows you to quickly check the syntax of a program without incurring the overhead of code generation. The **.lst** file is still produced, so you can get diagnostic information to begin debugging.

After fixing any program errors, you can change back to the default (**-qobject**) to test whether the program works correctly. If it does not work correctly, compile with the **-g** option for interactive debugging.

Restrictions

The **-qhalt** option can override the **-qobject** option, and **-qnoobject** can override **-qhalt**.

Related Information

See “Options That Control Listings and Messages” on page 61 and “Object Section” on page 337.

“The Compiler Phases” on page 351 gives some technical information about the compiler phases.

-qonetrip Option

Syntax

-qonetrip | -qnoonetrip
ONETRIP | NOONETRIP

This is the long form of the “-1 Option” on page 75.

-qoptimize Option

Syntax

`-qOPTimize[=level] | -qNOOPTimize`
`OPTimize[(level)] | NOOPTimize`

This is the long form of the “-O Option” on page 88.

-qpdf Option

Syntax

-qpdf{1|2}

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.

To use PDF, follow these steps:

1. Compile some or all of the source files in a program with the **-qpdf1** option. You need to specify the **-O2** option, or preferably the **-O3**, **-O4**, or **-O5** option, for optimization. Pay special attention to the compiler options that you use to compile the files, because you will need to use the same options later.

In a large application, concentrate on those areas of the code that can benefit most from optimization. You do not need to compile all of the application's code with the **-qpdf1** option.

2. Run the program all the way through using a typical data set. The program records profiling information when it finishes. You can run the program multiple times with different data sets, and the profiling information is accumulated to provide an accurate count of how often branches are taken and blocks of code are executed.

Important: Use data that is representative of the data that will be used during a normal run of your finished program.

3. Relink your program using the same compiler options as before, but change **-qpdf1** to **-qpdf2**. Remember that **-L**, **-l**, and some others are linker options, and you can change them at this point. In this second compilation, the accumulated profiling information is used to fine-tune the optimizations. The resulting program contains no profiling overhead and runs at full speed.

For best performance, use the **-O3**, **-O4**, or **-O5** option with all compilations when you use PDF (as in the example above). If your application contains C or C++ code compiled with IBM XL C/C+ compilers, you can achieve additional PDF optimization by specifying the **-qpdf1** and **-qpdf2** options available on those compilers. Combining **-qpdf1**/**-qpdf2** and **-qipa** or **-O5** options (that is, link with IPA) on all Fortran and C/C++ code will lead to maximum PDF information being available for optimization.

Rules

The profile is placed in the current working directory or in the directory that the **PDFDIR** environment variable names, if that variable is set.

To avoid wasting compilation and execution time, make sure that the **PDFDIR** environment variable is set to an absolute path. Otherwise, you might run the application from the wrong directory, and it will not be able to locate the profile data files. When that happens, the program may not be optimized correctly or may be stopped by a segmentation fault. A segmentation fault might also happen if you change the value of the **PDFDIR** variable and execute the application before finishing the PDF process.

Background Information

Because this option requires compiling the entire application twice, it is intended to be used after other debugging and tuning is finished, as one of the last steps before putting the application into production.

Restrictions

- PDF optimizations also require at least the **-O2** optimization level.
- You must compile the main program with PDF for profiling information to be collected at run time.
- Do not compile or run two different applications that use the same **PDFDIR** directory at the same time, unless you have used the **-qipa=pdfname** suboption to distinguish the sets of profiling information.
- You must use the same set of compiler options at all compilation steps for a particular program. Otherwise, PDF cannot optimize your program correctly and may even slow it down. All compiler settings must be the same, including any supplied by configuration files.
- If **-qipa** is not invoked either directly or through other options, **-qpdf1** and **-qpdf2** will invoke the **-qipa=level=0** option.
- If you do compile a program with **-qpdf1**, remember that it will generate profiling information when it runs, which involves some performance overhead. This overhead goes away when you recompile with **-qpdf2** or with no PDF at all.

The following commands, in the directory **/opt/ibmcmp/xlf/9.1/bin**, are available for managing the **PDFDIR** directory:

cleanpdf [*pathname*] Removes all profiling information from the *pathname* directory; or if *pathname* is not specified, from the **PDFDIR** directory; or if **PDFDIR** is not set, from the current directory.

Removing the profiling information reduces the run-time overhead if you change the program and then go through the PDF process again.

Run this program after compiling with **-qpdf2** or after finishing with the PDF process for a particular application. If you continue using PDF with an application after running **cleanpdf**, you must recompile all the files with **-qpdf1**.

mergepdf	<p>Generates a single pdf record from 2 or more input pdf records. All pdf records must come from the same executable.</p> <p>mergepdf automatically scales each pdf record (that is, file) based on its "weight". A scaling ratio can be specified for each pdf record, so that more important training runs can be weighted heavier than less important ones. The syntax for mergepdf is:</p> <pre>mergepdf [-r1] record1 [-r2] record2 ... -outputrecname [-n] [-v]</pre> <p>where:</p> <ul style="list-style-type: none"> -r The scaling ratio for the pdf record. If -r is not specified for an input record, the default ratio is 1.0 and no external scaling is applied. The scaling ratio must be greater than or equal to zero, and can be a floating point number or an integer. record The input file, or the directory that contains the pdf profile. -o output_recordname The pdf output directory name, or a file name that mergepdf will write the merged pdf record to. If a directory is specified, it must exist before you run the command. -n Do not normalize the pdf records. By default, records are normalized based on an internally calculated ratio for each profile before applying any user-specified ratio with -r. When -n is specified, the pdf records are scaled by the user-specified ratio -r. If -r is not specified, the pdf records are not scaled at all. -v Verbose mode displays the internal and user-specified scaling ratio used.
resetpdf [<i>pathname</i>]	Same as cleanpdf [<i>pathname</i>], described above. This command is provided for compatibility with the previous version.
showpdf	Displays the call and block counts for all procedures executed in a program run. To use this command, you must first compile your application specifying both -qpdf1 and -qshowpdf compiler options

Examples

Here is a simple example:

```
# Set the PDFDIR variable.
export PDFDIR=$HOME/project_dir
# Compile all files with -qpdf1.
xlf95 -qpdf1 -O3 file1.f file2.f file3.f
# Run with one set of input data.
a.out <sample.data
# Recompile all files with -qpdf2.
xlf95 -qpdf2 -O3 file1.f file2.f file3.f
# The program should now run faster than without PDF if
# the sample data is typical.
```

Here is a more elaborate example:

```
# Set the PDFDIR variable.
export PDFDIR=$HOME/project_dir
# Compile most of the files with -qpdf1.
xlf95 -qpdf1 -O3 -c file1.f file2.f file3.f
# This file is not so important to optimize.
xlf95 -c file4.f
# Non-PDF object files such as file4.o can be linked in.
xlf95 -qpdf1 file1.o file2.o file3.o file4.o
```

```
# Run several times with different input data.
a.out <polar_orbit.data
a.out <elliptical_orbit.data
a.out <geosynchronous_orbit.data
# Do not need to recompile the source of non-PDF object files (file4.f).
xlf95 -qpdf2 -O3 file1.f file2.f file3.f
# Link all the object files into the final application.
xlf95 file1.o file2.o file3.o file4.o
```

Related Information

See “XL Fortran Input Files” on page 24, “XL Fortran Output Files” on page 25, “Using Profile-directed Feedback (PDF)” on page 275, and “Optimizing Conditional Branching” on page 276.

-qphsinfo Option

Syntax

-qphsinfo | **-qnophsinfo**
PHSINFO | **NOPHSINFO**

The **-qphsinfo** compiler option displays timing information on the terminal for each compiler phase.

The output takes the form *number1/number2* for each phase where *number1* represents the CPU time used by the compiler and *number2* represents the total of the compiler time and the time that the CPU spends handling system calls.

Examples

To compile **app.f**, which consists of 3 compilation units, and report the time taken for each phase of the compilation, enter:

```
xlf90 app.f -qphsinfo
```

The output will look similar to:

```
FORTRAN phase 1 ftphas1      TIME = 0.000 / 0.000
** m_module   === End of Compilation 1 ===
FORTRAN phase 1 ftphas1      TIME = 0.000 / 0.000
** testassign === End of Compilation 2 ===
FORTRAN phase 1 ftphas1      TIME = 0.000 / 0.010
** dataassign === End of Compilation 3 ===
HOT           - Phase Ends; 0.000/ 0.000
HOT           - Phase Ends; 0.000/ 0.000
HOT           - Phase Ends; 0.000/ 0.000
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
W-TRANS      - Phase Ends; 0.000/ 0.000
OPTIMIZ      - Phase Ends; 0.000/ 0.000
REGALLO      - Phase Ends; 0.000/ 0.000
AS           - Phase Ends; 0.000/ 0.000
W-TRANS      - Phase Ends; 0.000/ 0.000
OPTIMIZ      - Phase Ends; 0.000/ 0.000
REGALLO      - Phase Ends; 0.000/ 0.000
AS           - Phase Ends; 0.000/ 0.000
1501-510     Compilation successful for file app.f.
```

Each phase is invoked three times, corresponding to each compilation unit. FORTRAN represents front-end parsing and semantic analysis, HOT loop transformations, W-TRANS intermediate language translation, OPTIMIZ high-level optimization, REGALLO register allocation and low-level optimization, and AS final assembly.

Compile **app.f** at the **-O4** optimization level with **-qphsinfo** specified:

```
xlf90 myprogram.f -qphsinfo -O4
```

The following output results:

```
FORTRAN phase 1 ftphas1      TIME = 0.010 / 0.020
** m_module   === End of Compilation 1 ===
FORTRAN phase 1 ftphas1      TIME = 0.000 / 0.000
** testassign === End of Compilation 2 ===
FORTRAN phase 1 ftphas1      TIME = 0.000 / 0.000
** dataassign === End of Compilation 3 ===
HOT          - Phase Ends; 0.000/ 0.000
HOT          - Phase Ends; 0.000/ 0.000
HOT          - Phase Ends; 0.000/ 0.000
IPA          - Phase Ends; 0.080/ 0.100
1501-510     Compilation successful for file app.f.
IPA          - Phase Ends; 0.050/ 0.070
W-TRANS     - Phase Ends; 0.010/ 0.030
OPTIMIZ     - Phase Ends; 0.020/ 0.020
REGALLO     - Phase Ends; 0.040/ 0.040
AS          - Phase Ends; 0.000/ 0.000
```

Note that during the IPA (interprocedural analysis) optimization phases, the program has resulted in one compilation unit; that is, all procedures have been inlined.

Related Information

“The Compiler Phases” on page 351.

-qpic Option

Syntax

`-qpic[=small|large]` | `-qnopic` (in 32-bit mode)
`-qpic[=small|large]` | `-qnopic` (in 64-bit mode)

The `-qpic` compiler option generates Position Independent Code (PIC) that can be used in shared libraries.

Arguments

small | **large** The **small** suboption tells the compiler to assume that the size of the Global Offset Table be at most, 64K. The **large** suboption allows the size of the Global Offset Table to be larger than 64K. This suboption allows more addresses to be stored in the Global Offset Table. However, it does generate code that is usually larger than that generated by the **small** suboption.

In **64-bit** mode, `-qpic=small` is the default.

-qnopic The compiler does not generate Position Independent Code.

In **32-bit** mode, `-qnopic` is the default.

Related Information

For more information on linker options, refer to the man pages for the `ld` command.

-qport Option

Syntax

`-qport[=suboptions]` | `-qnoport`
`PORT[(suboptions)]` | `NOPORT`

The **-qport** compiler option increases flexibility when porting programs to XL Fortran, providing a number of options to accommodate other Fortran language extensions. A particular suboption will always function independently of other **-qport** and compiler options.

Arguments

hexint | **nohexint**

If you specify this option, typeless constant hexadecimal strings are converted to integers when passed as an actual argument to the **INT** intrinsic function. Typeless constant hexadecimal strings not passed as actual arguments to **INT** remain unaffected.

mod | **nomod** Specifying this option relaxes existing constraints on the **MOD** intrinsic function, allowing two arguments of the same data type parameter to be of different kind type parameters. The result will be of the same type as the argument, but with the larger kind type parameter value.

nullarg | **nonnullarg**

For an external or internal procedure reference, specifying this option causes the compiler to treat an empty argument, which is delimited by a left parenthesis and a comma, two commas, or a comma and a right parenthesis, as a null argument. This suboption has no effect if the argument list is empty.

Examples of empty arguments are:

```
call foo(,,z)
```

```
call foo(x,,z)
```

```
call foo(x,y,)
```

The following program includes a null argument.

Fortran program:

```
program nularg
  real(4) res/0.0/
  integer(4) rc
  integer(4), external :: add
  rc = add(%val(2), res, 3.14, 2.18,) ! The last argument is a
                                   ! null argument.

  if (rc == 0) then
    print *, "res = ", res
  else
    print *, "number of arguments is invalid."
  endif
end program
```

C program:

```
int add(int a, float *res, float *b, float *c, float *d)
{
  int ret = 0;
  if (a == 2)
    *res = *b + *c;
}
```

```

else if (a == 3)
  *res = (*b + *c + *d);
else
  ret = 1;
return (ret);
}

```

sce | **nosce** By default, the compiler performs short circuit evaluation in selected logical expressions using XL Fortran rules. Specifying **sce** allows the compiler to use non-XL Fortran rules. The compiler will perform short circuit evaluation if the current rules allow it.

typestmt | **notypestmt**

The TYPE statement, which behaves in a manner similar to the PRINT statement, is supported whenever this option is specified.

typlessarg | **notyplessarg**

Converts all typeless constants to default integers if the constants are actual arguments to an intrinsic procedure whose associated dummy arguments are of integer type. Dummy arguments associated with typeless actual arguments of noninteger type remain unaffected by this option.

Using this option may cause some intrinsic procedures to become mismatched in kinds. Specify **-qxlf77=intarg** to convert the kind to that of the longest argument.

Related Information

See the section on the **INT** and **MOD** intrinsic functions in the *XL Fortran Language Reference* for further information.

-qposition Option

Syntax

```
-qposition={appendold | appendunknown} ...  
POSITION({APPENDOLD | APPENDUNKNOWN} ...)
```

Positions the file pointer at the end of the file when data is written after an **OPEN** statement with no **POSITION=** specifier and the corresponding **STATUS=** value (**OLD** or **UNKNOWN**) is specified.

Rules

The position becomes **APPEND** when the first I/O operation moves the file pointer if that I/O operation is a **WRITE** or **PRINT** statement. If it is a **BACKSPACE**, **ENDFILE**, **READ**, or **REWIND** statement instead, the position is **REWIND**.

Examples

In the following example, **OPEN** statements that do not specify a **POSITION=** specifier, but specify **STATUS='old'** will open the file as if **POSITION='append'** was specified.

```
xlf95 -qposition=appendold opens_old_files.f
```

In the following example, **OPEN** statements that do not specify a **POSITION=** specifier, but specify **STATUS='unknown'** will open the file as if **POSITION='append'** was specified.

```
xlf95 -qposition=appendunknown opens_unknown_files.f
```

In the following example, **OPEN** statements that do not specify a **POSITION=** specifier, but specify either **STATUS='old'** or **STATUS='unknown'** will open the file as if **POSITION='append'** was specified.

```
xlf95 -qposition=appendold:appendunknown opens_many_files.f
```

Related Information

See “File Positioning” on page 287 and the section on the **OPEN** statement in the *XL Fortran Language Reference*.

-qprefetch Option

Syntax

-qprefetch | -qnoprefetch

Instructs the compiler to insert the prefetch instructions automatically where there are opportunities to improve code performance.

Related Information

For more information on prefetch directives, see **PREFETCH directives** in the *XL Fortran Language Reference* and *The POWER4 Processor Introduction and Tuning Guide*. To selectively control prefetch directives using trigger constants, see the “-qdirective Option” on page 120.

-qqcount Option

Syntax

`-qqcount` | `-qnoqcount`
`QCOUNT` | `NOQCOUNT`

Accepts the **Q** character-count edit descriptor (**Q**) as well as the extended-precision **Q** edit descriptor (**Qw.d**). With `-qnoqcount`, all **Q** edit descriptors are interpreted as the extended-precision **Q** edit descriptor.

Rules

The compiler interprets a **Q** edit descriptor as one or the other depending on its syntax and issues a warning if it cannot determine which one is specified.

Related Information

See *Q (Character Count) Editing* in the *XL Fortran Language Reference*.

-qrealsize Option

Syntax

`-qrealsize=bytes`
`REALSIZE(bytes)`

Sets the default size of **REAL**, **DOUBLE PRECISION**, **COMPLEX**, and **DOUBLE COMPLEX** values.

This option is intended for maintaining compatibility with code that is written for other systems. You may find it useful as an alternative to **-qautodbl** in some situations.

Rules

The option affects the sizes² of constants, variables, derived type components, and functions (which include intrinsic functions) for which no kind type parameter is specified. Objects that are declared with a kind type parameter or length, such as **REAL(4)** or **COMPLEX*16**, are not affected.

Arguments

The allowed values for *bytes* are:

- 4 (the default)
- 8

Results

This option determines the sizes of affected objects as follows:

Data Object	REALSIZE(4) in Effect	REALSIZE(8) in Effect
1.2	REAL(4)	REAL(8)
1.2e0	REAL(4)	REAL(8)
1.2d0	REAL(8)	REAL(16)
1.2q0	REAL(16)	REAL(16)
REAL	REAL(4)	REAL(8)
DOUBLE PRECISION	REAL(8)	REAL(16)
COMPLEX	COMPLEX(4)	COMPLEX(8)
DOUBLE COMPLEX	COMPLEX(8)	COMPLEX(16)

Similar rules apply to intrinsic functions:

- If an intrinsic function has no type declaration, its argument and return types may be changed by the **-qrealsize** setting.
- Any type declaration for an intrinsic function must agree with the default size of the return value.

This option is intended to allow you to port programs unchanged from systems that have different default sizes for data. For example, you might need **-qrealsize=8** for programs that are written for a CRAY computer. The default value of 4 for this option is suitable for programs that are written specifically for many 32-bit computers.

Setting **-qrealsize** to 8 overrides the setting of the **-qdpc** option.

2. In Fortran 90/95 terminology, these values are referred to as *kind type parameters*.

Examples

This example shows how changing the **-qrealsize** setting transforms some typical entities:

```
@PROCESS REALSIZE(8)
  REAL R                ! treated as a real(8)
  REAL(8) R8           ! treated as a real(8)
  DOUBLE PRECISION DP  ! treated as a real(16)
  DOUBLE COMPLEX DC    ! treated as a complex(16)
  COMPLEX(4) C         ! treated as a complex(4)
  PRINT *,DSIN(DP)     ! treated as qsin(real(16))
! Note: we cannot get dsin(r8) because dsin is being treated as qsin.
END
```

Specifying **-qrealsize=8** affects intrinsic functions, such as **DABS**, as follows:

```
INTRINSIC DABS          ! Argument and return type become REAL(16).
DOUBLE PRECISION DABS ! OK, because DOUBLE PRECISION = REAL(16)
                       ! with -qrealsize=8 in effect.
REAL(16) DABS          ! OK, the declaration agrees with the option setting.
REAL(8) DABS           ! The declaration does not agree with the option
                       ! setting and is ignored.
```

Related Information

“-qintsize Option” on page 147 is a similar option that affects integer and logical objects. “-qautodbl Option” on page 106 is related to **-qrealsize**, although you cannot combine the options. When the **-qautodbl** option turns on automatic doubling, padding, or both, the **-qrealsize** option has no effect.

Type Parameters and Specifiers in the *XL Fortran Language Reference* discusses kind type parameters.

-qrecur Option

Syntax

-qrecur | **-qnorecur**
RECUR | **NORECUR**

Not recommended. Specifies whether external subprograms may be called recursively. For new programs, use the **RECURSIVE** keyword, which provides a standard-conforming way of using recursive procedures. If you specify the **-qrecur** option, the compiler must assume that any procedure could be recursive. Code generation for recursive procedures may be less efficient. Using the **RECURSIVE** keyword allows you to specify exactly which procedures are recursive.

Examples

! The following RECUR recursive function:

```
@process recur
function factorial (n)
integer factorial
if (n .eq. 0) then
    factorial = 1
else
    factorial = n * factorial (n-1)
end if
end function factorial
```

! can be rewritten to use F90/F95 RECURSIVE/RESULT features:

```
recursive function factorial (n) result (res)
integer res
if (n .eq. 0) then
    res = 1
else
    res = n * factorial (n-1)
end if
end function factorial
```

Restrictions

If you use the **xl**, **xl_r**, **f77**, or **fort77** command to compile programs that contain recursive calls, specify **-qnosave** to make the default storage class automatic.

-qreport Option

Syntax

```
-qreport[={smplist | hotlist}...]  
-qnoreport  
REPORT[({SMPLIST | HOTLIST}...)] NOREPORT
```

Determines whether to produce transformation reports showing how the program is parallelized and how loops are optimized.

You can use the **smp`list`** suboption to debug or tune the performance of SMP programs by examining the low-level transformations. You can see how the program deals with data and the automatic parallelization of loops. Comments within the listing tell you how the transformed program corresponds to the original source code and include information as to why certain loops were not parallelized.

You can use the **hot`list`** suboption to generate a report showing how loops are transformed.

Arguments

smp`list`

Produces a pseudo-Fortran listing that shows how the program is parallelized. This listing is produced before loop and other optimizations are performed. It includes messages that point out places in the program that can be modified to be more efficient. This report will only be produced if the **-qsmp** option is in effect.

hot`list` Produces a pseudo-Fortran listing that shows how loops are transformed, to assist you in tuning the performance of all loops. This suboption is the default if you specify **-qreport** with no suboptions.

In addition, if you specify the **-qreport=hot`list`** option when the **-qsmp** option is in effect, a pseudo-Fortran listing will be produced that shows the calls to the SMP runtime and the procedures created for parallel constructs.

Background Information

The transformation listing is part of the compiler listing file.

Restrictions

Loop transformation and auto parallelization are done on the link step at a **-O5** (or **-qipa=level=2**) optimization level. The **-qreport** option will generate the report in the listing file on the link step.

You must specify the **-qsmp** or the **-qhot** option to generate a loop transformation listing. You must specify the **-qsmp** option to generate a parallel transformation listing or parallel performance messages.

The code that the listing shows is not intended to be compilable. Do not include any of this code in your own programs or explicitly call any of the internal routines whose names appear in the listing.

Examples

To produce a listing file that you can use to tune parallelization:

```
xlf_r -qsmp -O3 -qhot -qreport=smplist needs_tuning.f
```

To produce a listing file that you can use to tune both parallelization and loop performance:

```
xlf_r -qsmp -03 -qhot -qreport=smp1ist:hotlist needs_tuning.f
```

To produce a listing file that you can use to tune only the performance of loops:

```
xlf95_r -03 -qhot -qreport=hotlist needs_tuning.f
```

Related Information

See “-qpdf Option” on page 175.

-qsaa Option

Syntax

-qsaa | -qnosaa
SAA | NOSAA

Checks for conformance to the SAA FORTRAN language definition. It identifies nonconforming source code and also options that allow such nonconformances.

Rules

These warnings have a prefix of **(L)**, indicating a problem with the language level.

Restrictions

The **-qflag** option can override this option.

Related Information

Use the “-qlanglvl Option” on page 156 to check your code for conformance to international standards.

-qsave Option

Syntax

```
-qsave[={all|defaultinit}] | -qnosave  
SAVE[({all|defaultinit})] NOSAVE
```

This specifies the default storage class for local variables.

If **-qsave=all** is specified, the default storage class is **STATIC**; if **-qnosave** is specified, the default storage class is **AUTOMATIC**; if **-qsave=defaultinit** is specified, the default storage class is **STATIC** for variables of derived type that have default initialization specified, and **AUTOMATIC** otherwise. The default suboption for the **-qsave** option is **all**. The two suboptions are mutually exclusive.

The default for this option depends on the invocation used. For example, you may need to specify **-qsave** to duplicate the behavior of FORTRAN 77 programs. The **xlF**, **xlF_r**, **f77**, and **fort77** commands have **-qsave** listed as a default option in **/etc/opt/ibmcomp/xlF/9.1/xlF.cfg** to preserve the previous behavior.

The following example illustrates the impact of the **-qsave** option on derived data type:

```
PROGRAM P  
  CALL SUB  
  CALL SUB  
END PROGRAM P  
  
SUBROUTINE SUB  
  LOGICAL, SAVE :: FIRST_TIME = .TRUE.  
  STRUCTURE /S/  
    INTEGER I/17/  
  END STRUCTURE  
  RECORD /S/ LOCAL_STRUCT  
  INTEGER LOCAL_VAR  
  
  IF (FIRST_TIME) THEN  
    LOCAL_STRUCT.I = 13  
    LOCAL_VAR = 19  
    FIRST_TIME = .FALSE.  
  ELSE  
    ! Prints " 13" if compiled with -qsave or -qsave=all  
    ! Prints " 13" if compiled with -qsave=defaultinit  
    ! Prints " 17" if compiled with -qnosave  
    PRINT *, LOCAL_STRUCT  
    ! Prints " 19" if compiled with -qsave or -qsave=all  
    ! Value of LOCAL_VAR is undefined otherwise  
    PRINT *, LOCAL_VAR  
  END IF  
END SUBROUTINE SUB
```

Related Information

The **-qnosave** option is usually necessary for multi-threaded applications and subprograms that are compiled with the “-qrecur Option” on page 189.

See *Storage Classes for Variables* in the *XL Fortran Language Reference* for information on how this option affects the storage class of variables.

-qsaveopt Option

Syntax

-qsaveopt | -qnosaveopt

Saves the command-line options used for compiling a source file, and other information, in the corresponding object file. The compilation must produce an object file for this option to take effect. Only one copy of the command-line options is saved, even though each object may contain multiple compilation units.

To list the options used, issue the **strings -a** command on the object file. The following is listed:

```
opt source_type invocation_used compilation_options
```

For example, if the object file is **t.o**, the **strings -a t.o** command may produce information similar to the following:

```
@(#) opt f /opt/ibmcmp/xlf/9.1/bin/xlf90 -qlist -qsaveopt t.f
```

where **f** identifies the source used as Fortran, **/opt/ibmcmp/xlf/9.1/bin/xlf90** shows the invocation command used, and **-qlist -qsaveopt** shows the compilation options.

-qsclk Option

Syntax

`-qsclk[=centi | micro]`

Specifies the resolution that the **SYSTEM_CLOCK** intrinsic procedure uses in a program. The default is centisecond resolution (`-qsclk=centi`). To use microsecond resolution, specify `-qsclk=micro`.

Related Information

See **SYSTEM_CLOCK** in the *XL Fortran Language Reference* for more information on returning integer data from a real-time clock.

-qshowpdf Option

Syntax

`-qshowpdf` | `-qnshowpdf`

Used together with `-qpdf1` to add additional call and block count profiling information to an executable.

When specified together with `-qpdf1`, the compiler inserts additional profiling information into the compiled application to collect call and block counts for all procedures in the application. Running the compiled application will record the call and block counts to the `._pdf` file.

After you run your application with training data, you can retrieve the contents of the `._pdf` file with the `showpdf` utility. This utility is described in “`-qpdf Option`” on page 175.

-qsigtrap Option

Syntax

`-qsigtrap[=trap_handler]`

When you are compiling a file that contains a main program, this option sets up the specified trap handler to catch **SIGTRAP** and **SIGFPE** exceptions. This option enables you to install a handler for **SIGTRAP** or **SIGFPE** signals without calling the **SIGNAL** subprogram in the program.

Arguments

To enable the `xl__trce` trap handler, specify **-qsigtrap** without a handler name. To use a different trap handler, specify its name with the **-qsigtrap** option.

If you specify a different handler, ensure that the object module that contains it is linked with the program.

Related Information

The possible causes of exceptions are described in “XL Fortran Run-Time Exceptions” on page 49. “Detecting and Trapping Floating-Point Exceptions” on page 257 describes a number of methods for dealing with exceptions that result from floating-point computations. “Installing an Exception Handler” on page 258 lists the exception handlers that XL Fortran supplies.

-qsmallstack Option

Syntax

`-qsmallstack[=dynlenonheap]`
| `-qnosmallstack`

Specifies that the compiler will minimize stack usage where possible.

The **-qsmallstack=dynlenonheap** suboption affects automatic objects that have nonconstant character lengths or a nonconstant array bound. When specified, automatic variables are allocated on the heap. When this suboption is not specified, automatic variables are allocated on the stack.

-qsmp Option

Syntax

-qsmp[=*suboptions*]

-qnosmp

Indicates that code should be produced for an SMP system. The default is to produce code for a uniprocessor machine. When you specify this option, the compiler recognizes all directives with the trigger constants **SMP\$**, **\$OMP**, and **IBMP** (unless you specify the **omp** suboption).

Only the **xlfr_r**, **xlfr90_r**, and **xlfr95_r** invocation commands automatically link in all of the thread-safe components. You can use the **-qsmp** option with the **xlfr**, **xlfr90**, **xlfr95**, **f77**, and **fort77** 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.

Arguments

auto | **noauto**

This suboption controls automatic parallelization. By default, the compiler will attempt to parallelize explicitly coded **DO** loops as well as those that are generated by the compiler for array language. If you specify the suboption **noauto**, automatic parallelization is turned off, and only constructs that are marked with prescriptive directives are parallelized. If the compiler encounters the **omp** suboption and the **-qsmp** or **-qsmp=auto** suboptions are not explicitly specified on the command line, the **noauto** suboption is implied. Also, note that **-qsmp=noopt** implies **-qsmp=noauto**. No automatic parallelization occurs under **-qsmp=noopt**; only user-directed parallelization will occur.

nested_par | nonested_par

If you specify the **nested_par** suboption, the compiler parallelizes prescriptive nested parallel constructs (**PARALLEL DO**, **PARALLEL SECTIONS**). 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. By default, the compiler serializes a nested parallel construct. Note that this option 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.

Note that the implementation of the **nested_par** suboption does not comply with the OpenMP Fortran API. If you specify this suboption, the run-time library uses the same threads for the nested **PARALLEL DO** and **PARALLEL SECTIONS** constructs that it used for the enclosing **PARALLEL** constructs.

omp | noomp

If you specify **-qsmp=omp**, the compiler enforces compliance with the OpenMP Fortran API. Specifying this option has the following effects:

- Automatic parallelization is turned off.
- All previously recognized directive triggers are ignored.

- The **-qcclines** compiler option is turned on if you specify **-qsmp=omp**.
- The **-qcclines** compiler option is not turned on if you specify **-qnocclines** and **-qsmp=omp**.
- The only recognized directive trigger is **\$OMP**. However, you can specify additional triggers on subsequent **-qdirective** options.
- The compiler issues warning messages if your code contains any language constructs that do not conform to the OpenMP Fortran API.

Specifying this option when the C preprocessor is invoked also defines the **_OPENMP** C preprocessor macro automatically, with the value *200011*, which is useful in supporting conditional compilation. This macro is only defined when the C preprocessor is invoked.

See *Conditional Compilation* in the *Language Elements* section of the *XL Fortran Language Reference* for more information.

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

rec_locks | norec_locks This suboption specifies whether recursive locks are used to avoid problems associated with **CRITICAL** constructs. If you specify the **rec_locks** suboption, a thread can enter a **CRITICAL** construct from within the dynamic extent of another **CRITICAL** construct that has the same name. If you specify **norec_locks**, a deadlock would occur in such a situation.

The default is **norec_locks**, or regular locks.

schedule=option The **schedule** suboption can take any one of the following subsuboptions:

affinity[=n]
 The iterations of a loop are initially divided into *number_of_threads* 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* has not been 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.

dynamic[=*n*]

The iterations of a loop are divided into chunks containing *n* iterations each. If *n* has not been specified, then the chunks consist of **CEILING**(number_of_iterations / number_of_threads) iterations.

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

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

guided[=*n*]

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

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. Active threads are assigned chunks on a "first-come, first-do" basis.

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 you have not specified *n*, the chunks will contain **CEILING**(number_of_iterations / number_of_threads) iterations. Each thread is assigned one of these chunks. This is known as *block scheduling*.

If a thread is asleep and it has been assigned work, it will be awakened so that it may complete its work.

For more information on chunking algorithms and **SCHEDULE**, refer to the *Directives* section in the *XL Fortran Language Reference*.

threshold=*n*

Controls the amount of automatic loop parallelization that occurs. The value of *n* represents the lower limit allowed for parallelization of a loop, based on the level of "work" present in a loop. 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. If this suboption is not specified, the program will use the default value *n*=100.

Rules

- If you specify **-qsmp** more than once, the previous settings of all suboptions are preserved, unless overridden by the subsequent suboption setting. The compiler does not override previous suboptions that you specify. The same is true for the version of **-qsmp** without suboptions; the default options are saved.

- Specifying the **omp** suboption always implies **noauto**, unless you specify **-qsmp** or **-qsmp=auto** on the command line.
- Specifying the **noomp** suboption always implies **auto**.
- The **omp** and **noomp** suboptions only appear in the compiler listing if you explicitly set them.
- If **-qsmp** is specified without any suboptions, **-qsmp=opt** becomes the default setting. If **-qsmp** is specified after the **-qsmp=noopt** suboption has been set, the **-qsmp=noopt** setting will always be ignored.
- If the option **-qsmp** with no suboptions follows the suboption **-qsmp=noopt** on a command line, the **-qsmp=opt** and **-qsmp=auto** options are enabled.
- Specifying the **-qsmp=noopt** suboption implies that **-qsmp=noauto**. It also implies **-qnoopt**. This option overrides performance options such as **-O2**, **-O3**, **-qhot**, anywhere on the command line unless **-qsmp** appears after **-qsmp=noopt**.
- 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.

Restrictions

The **-qsmp=noopt** suboption may affect the performance of the program.

Within the same **-qsmp** specification, you cannot specify the **omp** suboption before or after certain suboptions. The compiler issues warning messages if you attempt to specify them with **omp**:

auto This suboption controls automatic parallelization, but **omp** turns off automatic parallelization.

nested_par

Note that the implementation of the **nested_par** suboption does not comply with the OpenMP Fortran API. If you specify this suboption, the run-time library uses the same threads for the nested **PARALLEL DO** and **PARALLEL SECTIONS** constructs that it used for the enclosing **PARALLEL** constructs.

rec_locks

This suboption specifies a behaviour for **CRITICAL** constructs that is inconsistent with the OpenMP Fortran API.

schedule=affinity=*n*

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

Examples

The **-qsmp=noopt** suboption overrides performance optimization options anywhere on the command line unless **-qsmp** appears after **-qsmp=noopt**. The following examples illustrate that all optimization options that appear after **-qsmp=noopt** are processed according to the normal rules of scope and precedence.

Example 1

```
xlf90 -qsmp=noopt -O3...
is equivalent to
xlf90 -qsmp=noopt...
```

Example 2

```
    xlf90 -qsmp=noopt -O3 -qsmp...  
is equivalent to  
    xlf90 -qsmp -O3...
```

Example 3

```
    xlf90 -qsmp=noopt -O3 -qhot -qsmp -O2...  
is equivalent to  
    xlf90 -qsmp -qhot -O2...
```

If you specify the following, the compiler recognizes both the **\$OMP** and **SMP\$** directive triggers and issues a warning if a directive specified with either trigger is not allowed in OpenMP.

```
-qsmp=omp -qdirective=SMP$
```

If you specify the following, the **noauto** suboption is used. The compiler issues a warning message and ignores the **auto** suboption.

```
-qsmp=omp:auto
```

In the following example, you should specify **-qsmp=rec_locks** to avoid a deadlock caused by **CRITICAL** constructs.

```
program t  
  integer i, a, b  
  
  a = 0  
  b = 0  
!smp$ parallel do  
  do i=1, 10  
!smp$ critical  
  a = a + 1  
!smp$ critical  
  b = b + 1  
!smp$ end critical  
!smp$ end critical  
  enddo  
end
```

Related Information

If you use the **xlf**, **xlf_r**, **f77**, or **fort77** command with the **-qsmp** option to compile programs, specify **-qnosave** to make the default storage class automatic, and specify **-qthreaded** to tell the compiler to generate thread-safe code.

-qsource Option

Syntax

```
-qsource | -qnosource  
SOURCE | NOSOURCE
```

Determines whether to produce the source section of the listing.

This option displays on the terminal each source line where the compiler detects a problem, which can be very useful in diagnosing program errors in the Fortran source files.

You can selectively print parts of the source code by using **SOURCE** and **NOSOURCE** in **@PROCESS** directives in the source files around those portions of the program you want to print. This is the only situation where the **@PROCESS** directive does not have to be before the first statement of a compilation unit.

Examples

In the following example, the point at which the incorrect call is made is identified more clearly when the program is compiled with the **-qsource** option:

```
$ cat argument_mismatch.f  
    subroutine mult(x,y)  
    integer x,y  
    print *,x*y  
    end  
  
    program wrong_args  
    interface  
        subroutine mult(a,b) ! Specify the interface for this  
            integer a,b ! subroutine so that calls to it  
        end subroutine mult ! can be checked.  
    end interface  
    real i,j  
    i = 5.0  
    j = 6.0  
    call mult(i,j)  
    end  
  
$ xlf95 argument_mismatch.f  
** mult   === End of Compilation 1 ===  
"argument_mismatch.f", line 16.12: 1513-061 (S) Actual argument attributes  
do not match those specified by an accessible explicit interface.  
** wrong_args === End of Compilation 2 ===  
1501-511 Compilation failed for file argument_mismatch.f.  
$ xlf95 -qsource argument_mismatch.f  
** mult   === End of Compilation 1 ===  
    16 |   call mult(i,j)  
        .....a...  
a - 1513-061 (S) Actual argument attributes do not match those specified by  
an accessible explicit interface.  
** wrong_args === End of Compilation 2 ===  
1501-511 Compilation failed for file argument_mismatch.f.
```

Related Information

See "Options That Control Listings and Messages" on page 61 and "Source Section" on page 334.

-qspillsize Option

Syntax

`-qspillsize=bytes`
`SPILLSIZE(bytes)`

`-qspillsize` is the long form of `-NS`. See “`-N Option`” on page 87.

-qstrict Option

Syntax

```
-qstrict | -qnostrict  
STRICT | NOSTRICT
```

Ensures that optimizations done by default with the **-O3**, **-O4**, **-O5**, **-qhot**, and **-qipa** options, and optionally with the **-O2** option, do not alter the semantics of a program.

Defaults

For **-O3**, **-O4**, **-O5**, **-qhot**, and **-qipa**, the default is **-qnostrict**. For **-O2**, the default is **-qstrict**. This option is ignored for **-qnoopt**. With **-qnostrict**, optimizations may rearrange code so that results or exceptions are different from those of unoptimized programs.

This option is intended for situations where the changes in program execution in optimized programs produce different results from unoptimized programs. Such situations are likely rare because they involve relatively little-used rules for IEEE floating-point arithmetic.

Rules

With **-qnostrict** in effect, the following optimizations are turned on, unless **-qstrict** is also specified:

- Code that may cause an exception may be rearranged. The corresponding exception might happen at a different point in execution or might not occur at all. (The compiler still tries to minimize such situations.)
- Floating-point operations may not preserve the sign of a zero value. (To make certain that this sign is preserved, you also need to specify **-qfloat=rrm**, **-qfloat=nomaf**, or **-qfloat=strictnmaf**.)
- Floating-point expressions may be reassociated. For example, $(2.0*3.1)*4.2$ might become $2.0*(3.1*4.2)$ if that is faster, even though the result might not be identical.
- The **fltint** and **rsqrt** suboptions of the **-qfloat** option are turned on. You can turn them off again by also using the **-qstrict** option or the **nofltint** and **norsqrt** suboptions of **-qfloat**. With lower-level or no optimization specified, these suboptions are turned off by default.

Related Information

See “-O Option” on page 88, “-qhot Option” on page 140, and “-qfloat Option” on page 133.

-qstrictieemod Option

Syntax

`-qstrictieemod` | `-qnostrictieemod`
`STRICTIEEEMOD` | `NOSTRICTIEEEMOD`

Specifies whether the compiler will adhere to the draft Fortran 2003 IEEE arithmetic rules for the `ieee_arithmetic` and `ieee_exceptions` intrinsic modules. When you specify `-qstrictieemod`, the compiler adheres to the following rules:

- If there is an exception flag set on entry into a procedure that uses the IEEE intrinsic modules, the flag is set on exit. If a flag is clear on entry into a procedure that uses the IEEE intrinsic modules, the flag can be set on exit.
- If there is an exception flag set on entry into a procedure that uses the IEEE intrinsic modules, the flag clears on entry into the procedure and resets when returning from the procedure.
- When returning from a procedure that uses the IEEE intrinsic modules, the settings for halting mode and rounding mode return to the values they had at procedure entry.
- Calls to procedures that do not use the `ieee_arithmetic` or `ieee_exceptions` intrinsic modules from procedures that do use these modules, will not change the floating-point status except by setting exception flags.

Since the above rules can impact performance, specifying `-qnostrictieemod` will relax the rules on saving and restoring floating-point status. This prevents any associated impact on performance.

-qstrict_induction Option

Syntax

-qSTRICT_INDUCtion | -qNOSTRICT_INDUCtion

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.

You should avoid specifying **-qstrict_induction** unless absolutely necessary, as it may cause performance degradation.

Examples

Consider the following two examples:

Example 1

```
integer(1) :: i, j           ! Variable i can hold a
j = 0                       ! maximum value of 127.

do i = 1, 200               ! Integer overflow occurs when 128th
  j = j + 1                 ! iteration of loop is attempted.
enddo
```

Example 2

```
integer(1) :: i             ! Variable i can hold a maximum
i = 1_1                     ! value of 127.

100 continue
  if (i == -127) goto 200   ! Go to label 200 once decimal overflow
  i = i + 1_1              ! occurs and i == -127.
  goto 100
200 continue
  print *, i
end
```

If you compile these examples with the **-qstrict_induction** option, the compiler does not perform induction variable optimizations, but the performance of the code may be affected. If you compile the examples with the **-qnostrict_induction** option, the compiler may perform optimizations that may alter the semantics of the programs.

-qsuffix Option

Syntax

`-qsuffix=option=suffix`

Specifies the source-file suffix on the command line instead of in the `xlfcfg` file. This option saves time for the user by permitting files to be used as named with minimal makefile modifications and removes the risk of problems associated with modifying the `xlfcfg` file. Only one setting is supported at any one time for any particular file type.

Arguments

`f=suffix`

Where *suffix* represents the new *source-file-suffix*

`o=suffix`

Where *suffix* represents the new *object-file-suffix*

`s=suffix`

Where *suffix* represents the new *assembler-source-file-suffix*

`cpp=suffix`

Where *suffix* represents the new *preprocessor-source-file-suffix*

Rules

- The new suffix setting is case-sensitive.
- The new suffix can be of any length.
- Any setting for a new suffix will override the corresponding default setting in the `xlfcfg` file.
- If both `-qsuffix` and `-F` are specified, `-qsuffix` is processed last, so its setting will override the setting in the `xlfcfg` file.

Examples

For instance,

```
xlfc a.f90 -qsuffix=f=f90:cpp=F90
```

will cause these effects:

- The compiler is invoked for source files with a suffix of `.f90`.
- `cpp` is invoked for files with a suffix of `.F90`.

-qsuppress Option

Syntax

-qsuppress [=nnnn-mmm[:nnnn-mmm ...] | cmpmsg]

-qnosuppress

Arguments

nnnn-mmm[:nnnn-mmm ...]

Suppresses the display of a specific compiler message (*nnnn-mmm*) or a list of messages (*nnnn-mmm[:nnnn-mmm ...]*). *nnnn-mmm* is the message number. To suppress a list of messages, separate each message number with a colon.

cmpmsg

Suppresses the informational messages that report compilation progress and a successful completion.

This sub-option has no effect on any error messages that are emitted.

Background Information

In some situations, users may receive an overwhelming number of compiler messages. In many cases, these compiler messages contain important information. However, some messages contain information that is either redundant or can be safely ignored. When multiple error or warning messages appear during compilation, it can be very difficult to distinguish which messages should be noted. By using **-qsuppress**, you can eliminate messages that do not interest you.

- The compiler tracks the message numbers specified with **-qsuppress**. If the compiler subsequently generates one of those messages, it will not be displayed or entered into the listing.
- Only compiler and driver messages can be suppressed. Linker or operating system message numbers will be ignored if specified on the **-qextname** compiler option.
- If you are also specifying the **-qipa** compiler option, then **-qipa** must appear before the **-qextname** compiler option on the command line for IPA messages to be suppressed.

Restrictions

- The value of *nnnn* must be a four-digit integer between 1500 and 1585, since this is the range of XL Fortran message numbers.
- The value of *mmm* must be any three-digit integer (with leading zeros if necessary).

Examples

```
@process nullterm
  i = 1; j = 2;
  call printf("i=%d\n",%val(i));
  call printf("i=%d, j=%d\n",%val(i),%val(j));
end
```

Compiling this sample program would normally result in the following output:

```
"t.f", line 4.36: 1513-029 (W) The number of arguments to "printf" differ
from the number of arguments in a previous reference. You should use the
OPTIONAL attribute and an explicit interface to define a procedure with
optional arguments.
** _main    === End of Compilation 1 ===
1501-510  Compilation successful for file t.f.
```

When the program is compiled with **-qsuppress=1513-029**, the output is:

```
** _main    === End of Compilation 1 ===
1501-510  Compilation successful for file t.f.
```

Related Information

For another type of message suppression, see “-qflag Option” on page 132.

-qswapomp Option

Syntax

```
-qswapomp | -qnoswapomp  
SWAPOMP | NOSWAPOMP
```

Specifies that the compiler should recognize and substitute OpenMP routines in XL Fortran programs.

The OpenMP routines for Fortran and C have different interfaces. To support multi-language applications that use OpenMP routines, the compiler needs to recognize OpenMP routine names and substitute them with the XL Fortran versions of these routines, regardless of the existence of other implementations of such routines.

The compiler does not perform substitution of OpenMP routines when you specify the **-qnoswapomp** option.

Restrictions

The **-qswapomp** and **-qnoswapomp** options only affect Fortran sub-programs that reference OpenMP routines that exist in the program.

Rules

- If a call to an OpenMP routine resolves to a dummy procedure, module procedure, an internal procedure, a direct invocation of a procedure itself, or a statement function, the compiler will not perform the substitution.
- When you specify an OpenMP routine, the compiler substitutes the call to a different special routine depending upon the setting of the **-qintsize** option. In this manner, OpenMP routines are treated as generic intrinsic procedures.
- Unlike generic intrinsic procedures, if you specify an OpenMP routine in an **EXTERNAL** statement, the compiler will not treat the name as a user-defined external procedure. Instead, the compiler will still substitute the call to a special routine depending upon the setting of the **-qintsize** option.
- An OpenMP routine cannot be extended or redefined, unlike generic intrinsic procedures.

Examples

In the following example, the OpenMP routines are declared in an **INTERFACE** statement.

```
@PROCESS SWAPOMP  
  
INTERFACE  
  FUNCTION OMP_GET_THREAD_NUM()  
    INTEGER OMP_GET_THREAD_NUM  
  END FUNCTION OMP_GET_THREAD_NUM  
  
  FUNCTION OMP_GET_NUM_THREADS()  
    INTEGER OMP_GET_NUM_THREADS  
  END FUNCTION OMP_GET_NUM_THREADS  
END INTERFACE  
  
IAM = OMP_GET_THREAD_NUM()  
NP = OMP_GET_NUM_THREADS()  
PRINT *, IAM, NP  
END
```

Related Information

See the *OpenMP Execution Environment Routines and Lock Routines* section in the *XL Fortran Language Reference*.

-qtbtable Option

Syntax

`-qtbtable={none | small | full}`

Note: Applies to the 64-bit environment only.

Limits the amount of debugging traceback information in object files, which reduces the size of the program.

You can use this option to make your program smaller, at the cost of making it harder to debug. When you reach the production stage and want to produce a program that is as compact as possible, you can specify **-qtbtable=none**. Otherwise, the usual defaults apply: code compiled with **-g** or without **-O** has full traceback information (**-qtbtable=full**), and code compiled with **-O** contains less (**-qtbtable=small**).

Arguments

- none** The object code contains no traceback information at all. You cannot debug the program, because a debugger or other code-examination tool cannot unwind the program's stack at run time. If the program stops because of a run-time exception, it does not explain where the exception occurred.
- small** The object code contains traceback information but not the names of procedures or information about procedure parameters. You can debug the program, but some non-essential information is unavailable to the debugger. If the program stops because of a run-time exception, it explains where the exception occurred but reports machine addresses rather than procedure names.
- full** The object code contains full traceback information. The program is debuggable, and if it stops because of a run-time exception, it produces a traceback listing that includes the names of all procedures in the call chain.

Background Information

This option is most suitable for programs that contain many long procedure names, such as the internal names constructed for module procedures. You may find it more applicable to C++ programs than to Fortran programs.

Related Information

See “-g Option” on page 82, “-O Option” on page 88, “Debugging Optimized Code” on page 281, and “-qcompact Option” on page 114.

-qthreaded Option

Syntax

`-qthreaded`

Used by the compiler to determine when it must generate thread-safe code.

The **-qthreaded** option does not imply the **-qnosave** option. The **-qnosave** option specifies a default storage class of automatic for user local variables. In general, both of these options need to be used to generate thread-safe code. Simply specifying these options does not guarantee that your program is thread-safe. You should implement the appropriate locking mechanisms, as well.

Defaults

-qthreaded is the default for the `xlf90_r`, `xlf95_r`, and `xlf_r` commands.

Specifying the **-qthreaded** option implies **-qdirective=ibmt**, and by default, the *trigger_constant* **IBMT** is recognized.

-qtune Option

Syntax

`-qtune=implementation`

Tunes instruction selection, scheduling, and other implementation-dependent performance enhancements for a specific implementation of a hardware architecture. The compiler will use a **-qtune** setting that is compatible with the target architecture, which is controlled by the **-qarch**, **-q32**, and **-q64** options.

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.

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.

Arguments

Note: For Y-HPC, only **-qtune=auto** and **-qtune=ppc970** are valid choices for *implementation*; **-qtune=ppc970** is the default.

Otherwise, the choices are:

auto	Automatically detects the specific processor type of the compiling machine. It assumes that the execution environment will be the same as the compilation environment.
rs64b	The optimizations are tuned for the RS64II processor.
rs64c	The optimizations are tuned for the RS64III processor.
pwr3	The optimizations are tuned for the POWER3 processors.
pwr4	The optimizations are tuned for the POWER4 processors.
pwr5	The optimizations are tuned for the POWER5 processors.
ppc970	The optimizations are tuned for the PowerPC 970 processors.

If you do not specify **-qtune**, its setting is determined by the setting of the **-qarch** option, as follows:

-qarch Setting	Allowed -qtune Settings	Default -qtune Setting
ppc	See the list of acceptable -qtune settings under the -qarch=ppc64 entry.	pwr4
ppcgr	See the list of acceptable -qtune settings under the -qarch=ppc64gr entry.	pwr4
ppc64	rs64b, rs64c, pwr3, pwr4, pwr5, ppc970, auto	pwr4

-qarch Setting	Allowed -qtune Settings	Default -qtune Setting
ppc64gr	rs64b, rs64c, pwr3, pwr4, pwr5, ppc970, auto	pwr4
ppc64grsq	rs64b, rs64c, pwr3, pwr4, pwr5, ppc970, auto	pwr4
rs64b	rs64b, auto	rs64b
rs64c	rs64c, auto	rs64c
pwr3	pwr3, pwr4, pwr5, ppc970, auto	pwr3
pwr4	pwr4, pwr5, ppc970, auto	pwr4
pwr5	pwr5, auto	pwr5
ppc970	ppc970, auto	ppc970

Note that you can specify any **-qtune** suboption with **-qarch=auto** as long as you are compiling on a machine that is compatible with the **-qtune** suboption. For example, if you specify **-qarch=auto** and **-qtune=pwr5**, you must compile on a POWER3, POWER4, or POWER5 machine.

Related Information

See “-qarch Option” on page 100, “-qcache Option” on page 109, and “Compiling for Specific Architectures” on page 29.

-qundef Option

Syntax

`-qundef` | `-qnundef`
`UNDEF` | `NOUNDEF`

`-qundef` is the long form of the “-u Option” on page 236.

-qunroll Option

Syntax

`-qunroll[=auto | yes] | -qnounroll`

Specifies whether unrolling a **DO** loop is allowed in a program. Unrolling is allowed on outer and inner **DO** loops.

Arguments

- auto** The compiler performs basic loop unrolling. This is the default if **-qunroll** is not specified on the command line.
- yes** The compiler looks for more opportunities to perform loop unrolling than that performed with **-qunroll=auto**. Specifying **-qunroll** with no suboptions is equivalent to **-qunroll=yes**. In general, this suboption has more chances to increase compile time or program size than **-qunroll=auto** processing, but it may also improve your application's performance.

If you decide to unroll a loop, specifying one of the above suboptions does not automatically guarantee that the compiler will perform the operation. Based on the performance benefit, the compiler will determine whether unrolling will be beneficial to the program. Experienced compiler users should be able to determine the benefit in advance.

Rules

The **-qnounroll** option prohibits unrolling unless you specify the **STREAM_UNROLL**, **UNROLL**, or **UNROLL_AND_FUSE** directive for a particular loop. These directives always override the command line options.

Examples

In the following example, the **UNROLL(2)** directive is used to tell the compiler that the body of the loop can be replicated so that the work of two iterations is performed in a single iteration. Instead of performing 1000 iterations, if the compiler unrolls the loop, it will only perform 500 iterations.

```
!IBM* UNROLL(2)
DO I = 1, 1000
  A(I) = I
END DO
```

If the compiler chooses to unroll the previous loop, the compiler translates the loop so that it is essentially equivalent to the following:

```
DO I = 1, 1000, 2
  A(I) = I
  A(I+1) = I + 1
END DO
```

Related Information

See the appropriate directive on unrolling loops in the *XL Fortran Language Reference*.

- **STREAM_UNROLL**
- **UNROLL**
- **UNROLL_AND_FUSE**

See "Optimizing Loops and Array Language" on page 272.

-qunwind Option

Syntax

-qunwind		-qnounwind
<u>UNWIND</u>		NOUNWIND

Specifies that the compiler will preserve the default behavior for saves and restores to volatile registers during a procedure call. If you specify **-qnounwind**, the compiler rearranges subprograms to minimize saves and restores to volatile registers.

While code semantics are preserved, applications such as exception handlers that rely on the default behavior for saves and restores can produce undefined results. When using **-qnounwind** in conjunction with the **-g** compiler option, debug information regarding exception handling when unwinding the program's stack can be inaccurate.

-qversion Option

Syntax

`-qversion` | `-qnoversion`

Displays the version and release of the invoking compiler. Specify this option on its own with the compiler command. For example:

```
xlf90 -qversion
```

-qwarn64 Option

See “-qwarn64 Option” on page 247.

-qxflag=dvz Option

Syntax

`-qxflag=dvz`

Specifying **-qxflag=dvz** causes the compiler to generate code to detect floating-point divide-by-zero operations.

This option is only effective when you specify an optimization level of **-O** or higher.

With this option on, the extra code calls the external handler function `__xl_dzx` when the divisor is zero. The return value of this function is used as the result of the division. Users are required to provide the function to handle the divide-by-zero operations. Specifying **-qxflag=dvz** handles only single-precision (REAL*4) and double-precision (REAL*8) division.

The interface of the function is as follows:

```
real(8) function __xl_dzx(x, y, kind_type)
  real(8), value :: x, y
  integer, value :: kind_type
end function
```

where:

x is the dividend value

y is the divisor value

kind_type

specifies the size of the actual arguments associated with **x** and **y**.

A **kind_type** value equal to zero indicates that the actual arguments associated with **x** and **y** are of type REAL(8). A **kind_type** value equal to one indicates that the actual arguments associated with **x** and **y** are of type REAL(4).

The division always executes before the handler routine is called. This means that any exception is posted and handled before the handler function is called.

Related Information

See “XL Fortran Floating-Point Processing” on page 249, “-qfltrap Option” on page 135, and “Understanding XL Fortran Error Messages” on page 325.

-qxflag=oldtab Option

Syntax

`-qxflag=oldtab`
`XFLAG(OLDTAB)`

Interprets a tab in columns 1 to 5 as a single character (for fixed source form programs).

Defaults

By default, the compiler allows 66 significant characters on a source line after column 6. A tab in columns 1 through 5 is interpreted as the appropriate number of blanks to move the column counter past column 6. This default is convenient for those who follow the earlier Fortran practice of including line numbers or other data in columns 73 through 80.

Rules

If you specify the option `-qxflag=oldtab`, the source statement still starts immediately after the tab, but the tab character is treated as a single character for counting columns. This setting allows up to 71 characters of input, depending on where the tab character occurs.

-qxlf77 Option

Syntax

`-qxlf77=settings`
`XLf77(settings)`

Provides compatibility with FORTRAN 77 aspects of language semantics and I/O data format that have changed. Most of these changes are required by the Fortran 90 standard.

Defaults

By default, the compiler uses settings that apply to Fortran 95, Fortran 90, and the most recent compiler version in all cases; the default suboptions are **blankpad**, **nogedit77**, **nointarg**, **nointxor**, **leadzero**, **nooldboz**, **nopersistent**, and **nosofteof**. However, these defaults are only used by the `xl95`, `xl95_r`, `xl90`, `xl90_r`, `f90`, and `f95` commands, which you should use to compile new programs.

If you only want to compile and run old programs unchanged, you can continue to use the appropriate invocation command and not concern yourself with this option. You should only use this option if you are using existing source or data files with Fortran 90 or Fortran 95 and the `xl90`, `xl90_r`, `xl95`, `xl95_r`, `f90`, or `f95` command and find some incompatibility because of behavior or data format that has changed. Eventually, you should be able to recreate the data files or modify the source files to remove the dependency on the old behavior.

Arguments

To get various aspects of XL Fortran Version 2 behavior, select the nondefault choice for one or more of the following suboptions. The descriptions explain what happens when you specify the nondefault choices.

blankpad | **noblankpad**

For internal, direct-access, and stream-access files, uses a default setting equivalent to **pad='no'**. This setting produces conversion errors when reading from such a file if the format requires more characters than the record has. This suboption does not affect direct-access or stream-access files opened with a **pad=** specifier.

gedit77 | **nogedit77**

Uses FORTRAN 77 semantics for the output of **REAL** objects with the **G** edit descriptor. Between FORTRAN 77 and Fortran 90, the representation of 0 for a list item in a formatted output statement changed, as did the rounding method, leading to different output for some combinations of values and **G** edit descriptors.

intarg | **nointarg**

Converts all integer arguments of an intrinsic procedure to the kind of the longest argument if they are of different kinds. Under Fortran 90/95 rules, some intrinsics (for example, **IBSET**) determine the result type based on the kind of the first argument; others (for example, **MIN** and **MAX**) require that all arguments be of the same kind.

intxor | **nointxor**

Treats **.XOR.** as a logical binary intrinsic operator. It has a precedence equivalent to the **.EQV.** and **.NEQV.** operators and can be extended with an operator interface. (Because the semantics of **.XOR.** are identical to those of **.NEQV.**, **.XOR.** does not appear in the Fortran 90 or Fortran 95 language standard.)

Otherwise, the **.XOR.** operator is only recognized as a defined operator. The intrinsic operation is not accessible, and the precedence depends on whether the operator is used in a unary or binary context.

leadzero | **noleadzero**

Produces a leading zero in real output under the **D**, **E**, **L**, **F**, and **Q** edit descriptors.

oldboz | **nooldboz**

Turns blanks into zeros for data read by **B**, **O**, and **Z** edit descriptors, regardless of the **BLANK=** specifier or any **BN** or **BZ** control edit descriptors. It also preserves leading zeros and truncation of too-long output, which is not part of the Fortran 90 or Fortran 95 standard.

persistent | **nopersistent**

Saves the addresses of arguments to subprograms with **ENTRY** statements in static storage. This is an implementation choice that has been changed for increased performance.

softeof | **nosofteof**

Allows **READ** and **WRITE** operations when a unit is positioned after its endfile record unless that position is the result of executing an **ENDFILE** statement. This suboption reproduces a FORTRAN 77 extension of earlier versions of XL Fortran that some existing programs rely on.

-qxlf90 Option

Syntax

`-qxlf90={settings}`
`XLf90({settings})`

Provides compatibility with the Fortran 90 standard for certain aspects of the Fortran language.

Defaults

The default suboptions for `-qxlf90` depend on the invocation command that you specify. For the `xlf95` or `xlf95_r` command, the default suboptions are **signedzero** and **autodealloc**. For all other invocation commands, the defaults are **nosignedzero** and **noautodealloc**.

Arguments

signedzero | **nosignedzero**

Determines how the **SIGN(A,B)** function handles signed real 0.0. If you specify the `-qxlf90=signedzero` compiler option, **SIGN(A,B)** returns `-|A|` when `B=-0.0`. This behavior conforms to the Fortran 95 standard and is consistent with the IEEE standard for binary floating-point arithmetic. Note that for the **REAL(16)** data type, XL Fortran never treats zero as negative zero.

This suboption also determines whether a minus sign is printed in the following cases:

- For a negative zero in formatted output. Again, note that for the **REAL(16)** data type, XL Fortran never treats zero as negative zero.
- For negative values that have an output form of zero (that is, where trailing non-zero digits are truncated from the output so that the resulting output looks like zero). Note that in this case, the **signedzero** suboption does affect the **REAL(16)** data type; non-zero negative values that have an output form of zero will be printed with a minus sign.

autodealloc | **noautodealloc**

Determines whether the compiler deallocates allocatable objects that are declared locally without either the **SAVE** or the **STATIC** attribute and have a status of currently allocated when the subprogram terminates. This behavior conforms with the Fortran 95 standard. If you are certain that you are deallocating all local allocatable objects explicitly, you may wish to turn off this suboption to avoid possible performance degradation.

Examples

Consider the following program:

```
PROGRAM TESTSIGN
REAL X, Y, Z
X=1.0
Y=-0.0
Z=SIGN(X,Y)
PRINT *,Z
END PROGRAM TESTSIGN
```

The output from this example depends on the invocation command and the **-qxf90** suboption that you specify. For example:

Invocation Command/xlf90 Suboption	Output
xlf95	-1.0
xlf95 -qxf90=signedzero	-1.0
xlf95 -qxf90=nosignedzero	1.0
xlf90	1.0
xlf	1.0

Related Information

See the **SIGN** information in the *Intrinsic Procedures* section and the *Arrays Concepts* section of the *XL Fortran Language Reference*.

-qxlines Option

Syntax

```
-qxlines | -qnoxlines  
XLINES | NOXLINES
```

Specifies whether fixed source form lines with a X in column 1 are compiled or treated as comments. This option is similar to the recognition of the character 'd' in column 1 as a conditional compilation (debug) character. The **-D** option recognizes the character 'x' in column 1 as a conditional compilation character when this compiler option is enabled. The 'x' in column 1 is interpreted as a blank, and the line is handled as source code.

Defaults

This option is set to **-qnoxlines** by default, and lines with the character 'x' in column 1 in fixed source form are treated as comment lines. While the **-qxlines** option is independent of **-D**, all rules for debug lines that apply to using 'd' as the conditional compilation character also apply to the conditional compilation character 'x'. The **-qxlines** compiler option is only applicable to fixed source form.

The conditional compilation characters 'x' and 'd' may be mixed both within a fixed source form program and within a continued source line. If a conditional compilation line is continued onto the next line, all the continuation lines must have 'x' or 'd' in column 1. If the initial line of a continued compilation statement is not a debugging line that begins with either 'x' or 'd' in column 1, subsequent continuation lines may be designated as debug lines as long as the statement is syntactically correct.

The OMP conditional compilation characters '!\$', 'C\$', and '*\$' may be mixed with the conditional characters 'x' and 'd' both in fixed source form and within a continued source line. The rules for OMP conditional characters will still apply in this instance.

Examples

An example of a base case of -qxlines:

```
C2345678901234567890  
    program p  
      i=3 ; j=4 ; k=5  
X    print *,i,j  
X  +      ,k  
      end program p  
  
<output>: 3 4 5      (if -qxlines is on)  
          no output (if -qxlines is off)
```

In this example, conditional compilation characters 'x' and 'd' are mixed, with 'x' on the initial line:

```
C2345678901234567890
      program p
      i=3 ; j=4 ; k=5
X     print *,i,
D     +         j,
X     +         k
      end program p

<output>: 3 4 5 (if both -qxlines and -qdlines are on)
          3 5  (if only -qxlines is turned on)
```

Here, conditional compilation characters 'x' and 'd' are mixed, with 'd' on the initial line:

```
C2345678901234567890
      program p
      i=3 ; j=4 ; k=5
D     print *,i,
X     +         j,
D     +         k
      end program p

<output>: 3 4 5 (if both -qxlines and -qdlines are on)
          3 5  (if only -qdlines is turned on)
```

In this example, the initial line is not a debug line, but the continuation line is interpreted as such, since it has an 'x' in column 1:

```
C2345678901234567890
      program p
      i=3 ; j=4 ; k=5
      print *,i
X     +         ,j
X     +         ,k
      end program p

<output>: 3 4 5 (if -qxlines is on)
          3    (if -qxlines is off)
```

Related Information

See “-D Option” on page 79 and *Conditional Compilation in the Language Elements* section of the *XL Fortran Language Reference*.

-qxref Option

Syntax

`-qxref[=full] | -qnoxref`
`XREF[(FULL)] | NOXREF`

Determines whether to produce the cross-reference component of the attribute and cross-reference section of the listing.

If you specify only **-qxref**, only identifiers that are used are reported. If you specify **-qxref=full**, the listing contains information about all identifiers that appear in the program, whether they are used or not.

If **-qxref** is specified after **-qxref=full**, the full cross-reference listing is still produced.

You can use the cross-reference listing during debugging to locate problems such as using a variable before defining it or entering the wrong name for a variable.

Related Information

See “Options That Control Listings and Messages” on page 61 and “Attribute and Cross-Reference Section” on page 336.

-qzerosize Option

Syntax

-qzerosize | **-qnozerosize**
ZEROSIZE | **NOZEROSIZE**

Improves performance of FORTRAN 77 and some Fortran 90 and Fortran 95 programs by preventing checking for zero-sized character strings and arrays.

For Fortran 90 and Fortran 95 programs that might process such objects, use **-qzerosize**. For FORTRAN 77 programs, where zero-sized objects are not allowed, or for Fortran 90 and Fortran 95 programs that do not use them, compiling with **-qnozerosize** can improve the performance of some array or character-string operations.

Defaults

The default setting depends on which command invokes the compiler: **-qzerosize** for the **xlF90**, **xlF90_r**, **xlF95**, **xlF95_r**, **f90**, and **f95** commands and **-qnozerosize** for the **xlF**, **xlF_r**, and **f77/fort77** commands (for compatibility with FORTRAN 77).

Rules

Run-time checking performed by the **-C** option takes slightly longer when **-qzerosize** is in effect.

-S Option

Syntax

-S

Produces one or more `.s` files that show equivalent assembler source for each Fortran source file.

Rules

When this option is specified, the compiler produces the assembler source files as output instead of an object or an executable file.

Restrictions

The generated assembler files do not include all the data that is included in a `.o` file by `-qipa` or `-g`.

Examples

```
xlf95 -O3 -qhot -S test.f           # Produces test.s
```

Related Information

The “-o Option” on page 90 can be used to specify a name for the resulting assembler source file.

For information about the assembler-language format, see the *Assembler Language Reference*.

-t Option

Syntax

-tcomponents

Applies the prefix specified by the **-B** option to the designated components. *components* can be one or more of **a**, **F**, **c**, **h**, **I**, **b**, **z**, **l**, or **d** with no separators, corresponding to the assembler, the C preprocessor, the compiler, the array language optimizer, the interprocedural analysis (IPA) tool/loop optimizer, the code generator, the binder, the linker, and the **-S** disassembler, respectively.

Rules

If **-t** is not specified, any **-B** prefix is applied to all components.

Component	-t Mnemonic	Standard Program Name
assembler	a	as
C preprocessor	F	cpp
compiler front end	c	xlffentry
array language optimizer	h	xlffhot
IPA/loop optimizer	I	ipa
code generator	b	xlffcode
binder	z	bolt
linker	l	ld
disassembler	d	dis

Related Information

See “-B Option” on page 76 (which includes an example).

-U Option

Syntax

`-U`
`MIXED` | `NOMIXED`

Makes the compiler sensitive to the case of letters in names.

You can use this option when writing mixed-language programs, because Fortran names are all lowercase by default, while names in C and other languages may be mixed-case.

Rules

If `-U` is specified, case is significant in names. For example, the names `Abc` and `ABC` refer to different objects.

The option changes the link names used to resolve calls between compilation units. It also affects the names of modules and thus the names of their `.mod` files.

Defaults

By default, the compiler interprets all names as if they were in lowercase. For example, `Abc` and `ABC` are both interpreted as `abc` and so refer to the same object.

Restrictions

The names of intrinsics must be all in lowercase when `-U` is in effect. Otherwise, the compiler may accept the names without errors, but the compiler considers them to be the names of external procedures, rather than intrinsics.

Related Information

This is the short form of `-qmixed`. See “`-qmixed` Option” on page 167.

-u Option

Syntax

-u
UNDEF | **NOUNDEF**

Specifies that no implicit typing of variable names is permitted. It has the same effect as using the **IMPLICIT NONE** statement in each scope that allows implicit statements.

Defaults

By default, implicit typing is allowed.

Related Information

See **IMPLICIT** in the *XL Fortran Language Reference*.

This is the short form of **-qundef**. See “**-qundef Option**” on page 218.

-v Option

Syntax

`-v`

Generates information on the progress of the compilation.

Rules

As the compiler executes commands to perform different compilation steps, this option displays a simulation of the commands it calls and the system argument lists it passes.

For a particular compilation, examining the output that this option produces can help you determine:

- What files are involved
- What options are in effect for each step
- How far a compilation gets when it fails

Related Information

“`-# Option`” on page 74 is similar to `-v`, but it does not actually execute any of the compilation steps.

-V Option

Syntax

`-V`

This option is the same as `-v` except that you can cut and paste directly from the display to create a command.

-W Option

Syntax

-Wcomponent,options

Passes the listed options to a component that is executed during compilation. *component* is one of **a**, **F**, **c**, **h**, **I**, **b**, **z**, **l**, or **d**, corresponding to the assembler, the C preprocessor, the compiler, the array language optimizer, the interprocedural analysis (IPA) tool/loop optimizer, the code generator, the binder, the linker, and the **-S** disassembler, respectively.

Component	-W Mnemonic	Standard Program Name
assembler	a	as
C preprocessor	F	cpp
compiler front end	c	xlfcntry
array language optimizer	h	xlshot
IPA/loop optimizer	I	ipa
code generator	b	xlfcde
binder	z	bol
linker	l	ld
disassembler	d	dis

In the string following the **-W** option, use a comma as the separator for each option, and do not include any spaces. For example:

```
-Wcomponent,option_1[option_2,...,option_n]
```

Background Information

The primary purpose of this option is to construct sequences of compiler options to pass to one of the optimizing preprocessors. It can also be used to fine-tune the link-edit step by passing parameters to the **ld** command.

Defaults

You do not need the **-W** option to pass most options to the linker: 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** (or the **ldopts** stanza in the configuration file).

If you need to include a character that is special to the shell in the option string, precede the character with a backslash.

Examples

See “Passing Command-Line Options to the “ld” or “as” Commands” on page 28.

You can use ****, to embed a literal comma in the string supplied to the **-W** option.

In the following example, the ****, embeds a literal comma in the **-WF** string and causes three arguments, rather than four, to be supplied to the C preprocessor.

```
$ xlf -qfree=f90 '-WF,-Dint1=1,-Dint2=2,-Dlist=3\,4' a.F
$ cat a.F
print *, int1
print *, int2
print *, list
end
```

The output from the program will be:

```
$ ./a.out  
1  
2  
3 4
```

-w Option

Syntax

`-w`

A synonym for the “-qflag Option” on page 132. It sets **-qflag=e:e**, suppressing warning and informational messages and also messages generated by language-level checking.

-y Option

Syntax

`-y{n | m | p | z}`
`IEEE(Near | Minus | Plus | Zero)`

Specifies the rounding mode for the compiler to use when evaluating constant floating-point expressions at compile time. It is equivalent to the `-qieee` option.

Arguments

n Round to nearest.
m Round toward minus infinity.
p Round toward plus infinity.
z Round toward zero.

Related Information

See “-O Option” on page 88 and “-qfloat Option” on page 133.

`-y` is the short form of “-qieee Option” on page 142.

Using XL Fortran in a 64-Bit Environment

The 64-bit environment addresses an increasing demand for larger storage requirements and greater processing power. The Linux operating system provides an environment that allows you to develop and execute programs that exploit 64-bit processors through the use of 64-bit pointers and 64-bit integers.

To support larger executables that can be fit within a 64-bit address space, a separate, 64-bit object form is used to meet the requirements of 64-bit executables. The linker binds 64-bit objects to create 64-bit executables. Note that objects that are bound together, statically or shared, must all be of the same object format. The following scenarios are not permitted and will fail to load, or execute, or both:

- A 64-bit object or executable that has references to symbols from a 32-bit library or shared library
- A 32-bit object or executable that has references to symbols from a 64-bit library or shared library
- A 64-bit executable that attempts to explicitly load a 32-bit module
- A 32-bit executable that attempts to explicitly load a 64-bit module
- Attempts to run 64-bit applications on 32-bit platforms

On both 64-bit and 32-bit platforms, 32-bit executables will continue to run as they currently do on a 32-bit platform.

The XL Fortran compiler mainly provides 64-bit mode support through the compiler option **-q64** in conjunction with the compiler option **-qarch**. This combination determines the bit mode and instruction set for the target architecture. The **-q32** and **-q64** options take precedence over the setting of the **-qarch** option. The **-q64** option will win over a 32-bit mode only **-qarch** setting, and the compiler will upgrade the **-qarch** setting to something that will handle 64-bit mode. Conflicts between the **-q32** and **-q64** options are resolved by the "last option wins" rule. Setting **-qarch=com** will ensure future compatibility for applications in 32-bit mode. For 64-bit mode applications, use **-qarch=ppc64** to achieve the same effect for all present or future supported 64-bit mode systems. **-qarch** settings that target a specific architecture, like the **rs64b**, **rs64c**, **pwr3**, **pwr4**, **pwr5**, **ppc970**, and **auto** settings will be more system-dependent.

Compiler Options for the 64-Bit Environment

The compiler options that are described in this section enable you to do the following:

- Develop applications for the 64-bit environment
- Help migrate source code from the 32-bit environment to a 64-bit environment

Some of these options already exist in the 32-bit environment but have new settings particular to the 64-bit architecture. This section only covers the new settings for these cases. The options that are grouped here are primarily for developers who are targeting 64-bit platforms.

-q32 Option

Syntax

-q32

Enables 32-bit compilation bit mode (or, more briefly, 32-bit mode) support in a 64-bit environment. The **-q32** option indicates the compilation bit mode and, together with the **-qarch** option, determines the target machines that the 32-bit executable will run on.

Rules

- The default integer and default real size are 4 bytes in 32-bit mode.
- The default integer pointer size is 4 bytes in 32-bit mode.
- 32-bit object modules are created when targeting 32-bit mode.
- **-q32** is the default.
- **-q64** may override **-q32**.
- All settings for **-qarch** are compatible with **-q32**. If you specify **-q32**, the default suboption is **ppc64grsq**, and the default **-qtune** suboption for **-q32** is **pwr4**.
- The **LOC** intrinsic returns an **INTEGER(4)** value.

Examples

- Using 32-bit compilation mode and targeting a generic PowerPC architecture:
-qarch=ppc -q32
- Now keep the same compilation mode, but change the target to RS64II:
-qarch=ppc -q32 -qarch=rs64b
Notice that the last setting for **-qarch** wins.
- Now keep the same target, but change the compilation mode to 64-bit:
-qarch=ppc -q32 -qarch=rs64b -q64

Notice that specifying **-q64** overrides the earlier instance of **-q32**.

-q64 Option

Syntax

-q64

Indicates the 64-bit compilation bit mode and, together with the **-qarch** option, determines the target machines on which the 64-bit executable will run. The **-q64** option indicates that the object module will be created in 64-bit object format and that the 64-bit instruction set will be generated. Note that you may compile in a 32-bit environment to create 64-bit objects, but you must link them in a 64-bit environment with the **-q64** option.

Rules

- Settings for **-qarch** that are compatible with **-q64** are as follows:
 - **-qarch=auto** (if compiling on a 64-bit system)
 - **-qarch=com** (With **-q64** and **-qarch=com**, the compiler will silently upgrade the arch setting to **ppc64**.)
 - **-qarch=ppc** (With **-q64** and **-qarch=ppc**, the compiler will silently upgrade the arch to **ppc64**.)
 - **-qarch=ppcgr** (With **-q64** and **-qarch=ppcgr**, the compiler will silently upgrade the arch to **ppc64gr**.)
 - **-qarch=ppc64**
 - **-qarch=ppc64gr**
 - **-qarch=ppc64grsq**
 - **-qarch=rs64b**
 - **-qarch=rs64c**
 - **-qarch=pwr3**
 - **-qarch=pwr4**
 - **-qarch=pwr5**
 - **-qarch=ppc970**
- The default **-qarch** setting for **-q64** is **ppc64**.
- 64-bit object modules are created when targeting 64-bit mode.
- **-q32** may override **-q64**.
- **-q64** will override a conflicting setting for **-qarch** and will result in the setting **-q64 -qarch=ppc64** along with a warning message.
- The default tune setting for **-q64** is **-qtune=pwr4**.
- The default integer and default real size is 4 bytes in 64-bit mode.
- The default integer pointer size is 8 bytes in 64-bit mode.
- The maximum array size increases to approximately $2^{*}40$ bytes (in static storage) or $2^{*}60$ bytes (in dynamic allocation on the heap). The maximum dimension bound range is extended to $-2^{*}63$, $2^{*}63-1$ bytes. The maximum array size for array constants has not been extended and will remain the same as the maximum in 32-bit mode. The maximum array size that you can initialize is $2^{*}28$ bytes.
- The maximum iteration count for array constructor implied DO loops increases to $2^{*}63-1$ bytes.
- The maximum character variable length extends to approximately $2^{*}40$ bytes. The maximum length of character constants and subobjects of constants remains the same as in 32-bit mode, which is 32 767 bytes (32 KB).
- The **LOC** intrinsic returns an **INTEGER(8)** value.
- When you use **-qautodbl=dblpad** in 64-bit mode, you should use **-qintsize=8** to promote **INTEGER(4)** to **INTEGER(8)** for 8 byte integer arithmetic.

Examples

This example targets the RS64II (also known as RS64b) in 64-bit mode:

```
-q32 -qarch=rs64b -q64
```

In this example 64-bit compilation that targets the common group of 64-bit architectures (which currently consists only of the RS64II, RS64III, POWER3, POWER4, POWER5, and PowerPC 970):

```
-q64 -qarch=com
```

arch setting is silently upgraded to **ppc64**, the most "common" 64-bit mode compilation target.

-qwarn64 Option

Syntax

`-qwarn64` | `-qnowarn64`

Aids in porting code from a 32-bit environment to a 64-bit environment by detecting the truncation of an 8-byte integer pointer to 4 bytes. The **-qwarn64** option uses informational messages to identify statements that may cause problems with the 32-bit to 64-bit migration.

Rules

- The default setting is **-qnowarn64**.
- You can use the **-qwarn64** option in both 32-bit and 64-bit modes.
- The compiler flags the following situations with informational messages:
 - The assignment of a reference to the **LOC** intrinsic to an **INTEGER(4)** variable.
 - The assignment between an **INTEGER(4)** variable or **INTEGER(4)** constant and an integer pointer.
 - The specification of an integer pointer within a common block.
 - The specification of an integer pointer within an equivalence statement.

XL Fortran Floating-Point Processing

This section answers some common questions about floating-point processing, such as:

- How can I get predictable, consistent results?
- How can I get the fastest or the most accurate results?
- How can I detect, and possibly recover from, exception conditions?
- Which compiler options can I use for floating-point calculations?

Related Information: This section makes frequent reference to the compiler options that are grouped together in “Options for Floating-Point Processing” on page 70, especially the “-qfloat Option” on page 133. The XL Fortran compiler also provides three intrinsic modules for exception handling and IEEE arithmetic support to help you write IEEE module-compliant code that can be more portable. See *IEEE Modules and Support* in the *XL Fortran Language Reference* for details.

The use of the compiler options for floating-point calculations affects the accuracy, performance, and possibly the correctness of floating-point calculations. Although the default values for the options were chosen to provide efficient and correct execution of most programs, you may need to specify nondefault options for your applications to work the way you want. We strongly advise you to read this section before using these options.

Note: The discussions of single-, double-, and extended-precision calculations in this section all refer to the default situation, with **-qrealsize=4** and no **-qautodbl** specified. If you change these settings, keep in mind that the size of a Fortran **REAL**, **DOUBLE PRECISION**, and so on may change, but single precision, double precision, and extended precision (in lowercase) still refer to 4-, 8-, and 16-byte entities respectively.

The information in this section relates to floating-point processing on the PowerPC family of processors.

IEEE Floating-Point Overview

Here is a brief summary of the *IEEE Standard for Floating-Point Arithmetic* and the details of how it applies to XL Fortran on specific hardware platforms. For information on the draft Fortran 2003 IEEE Module and arithmetic support, see the *XL Fortran Language Reference*.

Compiling for Strict IEEE Conformance

By default, XL Fortran follows most, but not all of the rules in the IEEE standard. To compile for strict compliance with the standard:

- Use the compiler option **-qfloat=nomaf**.
- If the program changes the rounding mode at run time, include **rrm** among the **-qfloat** suboptions.
- If the data or program code contains signaling NaN values (NaN), include **nans** among the **-qfloat** suboptions. (A signaling NaN is different from a quiet NaN; you must explicitly code it into the program or data or create it by using the **-qinitauto** compiler option.)

- If compiling with `-O3`, include the option `-qstrict` also.

IEEE Single- and Double-Precision Values

XL Fortran encodes single-precision and double-precision values in IEEE format. For the range and representation, see *Real* in the *XL Fortran Language Reference*.

IEEE Extended-Precision Values

The IEEE standard suggests, but does not mandate, a format for extended-precision values. XL Fortran does not use this format. “Extended-Precision Values” on page 253 describes the format that XL Fortran uses.

Infinities and NaNs

For single-precision real values:

- Positive infinity is represented by the bit pattern `X'7F80 0000'`.
- Negative infinity is represented by the bit pattern `X'FF80 0000'`.
- A signaling NaN is represented by any bit pattern between `X'7F80 0001'` and `X'7FBF FFFF'` or between `X'FF80 0001'` and `X'FFBF FFFF'`.
- A quiet NaN is represented by any bit pattern between `X'7FC0 0000'` and `X'7FFF FFFF'` or between `X'FFC0 0000'` and `X'FFFF FFFF'`.

For double-precision real values:

- Positive infinity is represented by the bit pattern `X'7FF00000 00000000'`.
- Negative infinity is represented by the bit pattern `X'FFF00000 00000000'`.
- A signaling NaN is represented by any bit pattern between `X'7FF00000 00000001'` and `X'7FF7FFFF FFFFFFFF'` or between `X'FFF00000 00000001'` and `X'FFF7FFFF FFFFFFFF'`.
- A quiet NaN is represented by any bit pattern between `X'7FF80000 00000000'` and `X'7FFFFFFF FFFFFFFF'` or between `X'FFF80000 00000000'` and `X'FFFFFFF FFFFFFFF'`.

These values do not correspond to any Fortran real constants. You can generate all of these by encoding the bit pattern directly, or by using the `ieee_value` function provided in the `ieee_arithmetic` module. Using the `ieee_value` function is the preferred programming technique, as it is allowed by the Fortran 2003 draft standard and the results are portable. Encoding the bit pattern directly could cause portability problems on machines using different bit patterns for the different values. All except signaling NaN values can occur as the result of arithmetic operations:

```

$ cat fp_values.f
real plus_inf, minus_inf, plus_nanq, minus_nanq, nans
real large

data plus_inf /z'7f800000'/
data minus_inf /z'ff800000'/
data plus_nanq /z'7fc00000'/
data minus_nanq /z'ffc00000'/
data nans /z'7f800001'/

print *, 'Special values:', plus_inf, minus_inf, plus_nanq, minus_nanq, nans

! They can also occur as the result of operations.
large = 10.0 ** 200
print *, 'Number too big for a REAL:', large * large
print *, 'Number divided by zero:', (-large) / 0.0
print *, 'Nonsensical results:', plus_inf - plus_inf, sqrt(-large)

! To find if something is a NaN, compare it to itself.
print *, 'Does a quiet NaN equal itself:', plus_nanq .eq. plus_nanq
print *, 'Does a signaling NaN equal itself:', nans .eq. nans
! Only for a NaN is this comparison false.

end
$ xlf95 -o fp_values fp_values.f
** _main === End of Compilation 1 ===
1501-510 Compilation successful for file fp_values.f.
$ fp_values
Special values: INF -INF NAN -NAN NAN
Number too big for a REAL: INF
Number divided by zero: -INF
Nonsensical results: NAN NAN
Does a quiet NaN equal itself: F
Does a signaling NaN equal itself: F

```

Exception-Handling Model

The IEEE standard defines several exception conditions that can occur:

OVERFLOW

The exponent of a value is too large to be represented.

UNDERFLOW

A nonzero value is so small that it cannot be represented without an extraordinary loss of accuracy. The value can be represented only as zero or a denormal number.

ZERODIVIDE

A finite nonzero value is divided by zero.

INVALID

Operations are performed on values for which the results are not defined. These include:

- Operations on signaling NaN values
- infinity - infinity
- 0.0 * infinity
- 0.0 / 0.0
- mod(x,y) or ieee_rem(x,y) (or other remainder functions) when x is infinite or y is zero
- The square root of a negative number
- Conversion of a floating point number to an integer when the converted value cannot be represented faithfully

- Comparisons involving NaN values

INEXACT

A computed value cannot be represented exactly, so a rounding error is introduced. (This exception is very common.)

XL Fortran always detects these exceptions when they occur, but the default is not to take any special action. Calculation continues, usually with a NaN or infinity value as the result. If you want to be automatically informed when an exception occurs, you can turn on exception trapping through compiler options or calls to intrinsic subprograms. However, different results, intended to be manipulated by exception handlers, are produced:

Table 14. Results of IEEE Exceptions, with and without Trapping Enabled

	Overflow	Underflow	Zerodivide	Invalid	Inexact
Exceptions not enabled (default)	INF	Denormalized number	INF	NaN	Rounded result
Exceptions enabled	Unnormalized number with biased exponent	Unnormalized number with biased exponent	No result	No result	Rounded result

Note: Because different results are possible, it is very important to make sure that any exceptions that are generated are handled correctly. See “Detecting and Trapping Floating-Point Exceptions” on page 257 for instructions on doing so.

Hardware-Specific Floating-Point Overview

Single- and Double-Precision Values

The PowerPC floating-point hardware performs calculations in either IEEE single-precision (equivalent to **REAL(4)** in Fortran programs) or IEEE double-precision (equivalent to **REAL(8)** in Fortran programs).

Keep the following considerations in mind:

- Double precision provides greater range (approximately $10^{**(-308)}$ to 10^{**308}) and precision (about 15 decimal digits) than single precision (approximate range $10^{**(-38)}$ to 10^{**38} , with about 7 decimal digits of precision).
- Computations that mix single and double operands are performed in double precision, which requires conversion of the single-precision operands to double-precision. These conversions do not affect performance.
- Double-precision values that are converted to single-precision (such as when you specify the **SNGL** intrinsic or when a double-precision computation result is stored into a single-precision variable) require rounding operations. A rounding operation produces the correct single-precision value, which is based on the IEEE rounding mode in effect. The value may be less precise than the original double-precision value, as a result of rounding error. Conversions from double-precision values to single-precision values may reduce the performance of your code.
- Programs that manipulate large amounts of floating-point data may run faster if they use **REAL(4)** rather than **REAL(8)** variables. (You need to ensure that **REAL(4)** variables provide you with acceptable range and precision.) The programs may run faster because the smaller data size reduces memory traffic, which can be a performance bottleneck for some applications.

The floating-point hardware also provides a special set of double-precision operations that multiply two numbers and add a third number to the product. These combined multiply-add (**MAF**) operations are performed at the same speed that either a multiply or an add operation alone is performed. The **MAF** functions provide an extension to the IEEE standard because they perform the multiply and add with one (rather than two) rounding errors. The **MAF** functions are faster and more accurate than the equivalent separate operations.

Extended-Precision Values

XL Fortran extended precision is not in the format suggested by the IEEE standard, which suggests extended formats using more bits in both the exponent (for greater range) and the fraction (for greater precision).

XL Fortran extended precision, equivalent to **REAL(16)** in Fortran programs, is implemented in software. Extended precision provides the same range as double precision (about $10^{**(-308)}$ to 10^{**308}) but more precision (a variable amount, about 31 decimal digits or more). The software support is restricted to round-to-nearest mode. Programs that use extended precision must ensure that this rounding mode is in effect when extended-precision calculations are performed. See “Selecting the Rounding Mode” on page 254 for the different ways you can control the rounding mode.

Programs that specify extended-precision values as hexadecimal, octal, binary, or Hollerith constants must follow these conventions:

- Extended-precision numbers are composed of two double-precision numbers with different magnitudes that do not overlap. That is, the binary exponents differ by at least the number of fraction bits in a **REAL(8)**. The high-order double-precision value (the one that comes first in storage) must have the larger magnitude. The value of the extended-precision number is the sum of the two double-precision values.
- For a value of NaN or infinity, you must encode one of these values within the high-order double-precision value. The low-order value is not significant.

Because an XL Fortran extended-precision value can be the sum of two values with greatly different exponents, leaving a number of assumed zeros in the fraction, the format actually has a variable precision with a minimum of about 31 decimal digits. You get more precision in cases where the exponents of the two double values differ in magnitude by more than the number of digits in a double-precision value. This encoding allows an efficient implementation intended for applications requiring more precision but no more range than double precision.

Notes:

1. In the discussions of rounding errors because of compile-time folding of expressions, keep in mind that this folding produces different results for extended-precision values more often than for other precisions.
2. Special numbers, such as NaN and infinity, are not fully supported for extended-precision values. Arithmetic operations do not necessarily propagate these numbers in extended precision.
3. XL Fortran does not always detect floating-point exception conditions (see “Detecting and Trapping Floating-Point Exceptions” on page 257) for extended-precision values. If you turn on floating-point exception trapping in programs that use extended precision, XL Fortran may also generate signals in cases where an exception condition does not really occur.

How XL Fortran Rounds Floating-Point Calculations

Understanding rounding operations in XL Fortran can help you get predictable, consistent results. It can also help you make informed decisions when you have to make tradeoffs between speed and accuracy.

In general, floating-point results from XL Fortran programs are more accurate than those from other implementations because of **MAF** operations and the higher precision used for intermediate results. If identical results are more important to you than the extra precision and performance of the XL Fortran defaults, read “Duplicating the Floating-Point Results of Other Systems” on page 257.

Selecting the Rounding Mode

To change the rounding mode in a program, you can call the **fpsets** and **fpgets** routines, which use an array of logicals named **fpstat**, defined in the include files **/opt/ibmcomp/xf/9.1/include/fpdt.h** and **fpdc.h**. The **fpstat** array elements correspond to the bits in the floating-point status and control register.

For floating-point rounding control, the array elements **fpstat(fprn1)** and **fpstat(fprn2)** are set as specified in the following table:

Table 15. Rounding-Mode Bits to Use with *fpsets* and *fpgets*

fpstat(fprn1)	fpstat(fprn2)	Rounding Mode Enabled
.true.	.true.	Round towards -infinity.
.true.	.false.	Round towards +infinity.
.false.	.true.	Round towards zero.
.false.	.false.	Round to nearest.

For example:

```
program fptest
  include 'fpdc.h'

  call fpgets( fpstat ) ! Get current register values.
  if ( (fpstat(fprn1) .eqv. .false.) .and. +
      (fpstat(fprn2) .eqv. .false.)) then
    print *, 'Before test: Rounding mode is towards nearest'
    print *, '          2.0 / 3.0 = ', 2.0 / 3.0
    print *, '          -2.0 / 3.0 = ', -2.0 / 3.0
  end if

  call fpgets( fpstat ) ! Get current register values.
  fpstat(fprn1) = .TRUE. ! These 2 lines mean round towards
  fpstat(fprn2) = .FALSE. ! +infinity.
  call fpsets( fpstat )
  r = 2.0 / 3.0
  print *, 'Round towards +infinity: 2.0 / 3.0= ', r

  call fpgets( fpstat ) ! Get current register values.
  fpstat(fprn1) = .TRUE. ! These 2 lines mean round towards
  fpstat(fprn2) = .TRUE. ! -infinity.
  call fpsets( fpstat )
  r = -2.0 / 3.0
  print *, 'Round towards -infinity: -2.0 / 3.0= ', r
end

! This block data program unit initializes the fpstat array, and so on.
```

```

        block data
        include 'fpdc.h'
        include 'fpdt.h'
    end

```

XL Fortran also provides several procedures that allow you to control the floating-point status and control register of the processor directly. These procedures are more efficient than the **fpsets** and **fpgets** subroutines because they are mapped into inlined machine instructions that manipulate the floating-point status and control register (fpscr) directly.

XL Fortran supplies the **get_round_mode()** and **set_round_mode()** procedures in the **xlf_fp_util** module. These procedures return and set the current floating-point rounding mode, respectively.

For example:

```

program fptest
    use, intrinsic :: xlf_fp_util
    integer(fpscr_kind) old_fpscr
    if ( get_round_mode() == fp_rnd_rn ) then
        print *, 'Before test: Rounding mode is towards nearest'
        print *, '          2.0 / 3.0 = ', 2.0 / 3.0
        print *, '          -2.0 / 3.0 = ', -2.0 / 3.0
    end if

    old_fpscr = set_round_mode( fp_rnd_rp )
    r = 2.0 / 3.0
    print *, 'Round towards +infinity: 2.0 / 3.0 = ', r

    old_fpscr = set_round_mode( fp_rnd_rm )
    r = -2.0 / 3.0
    print *, 'Round towards -infinity: -2.0 / 3.0 = ', r
end

```

XL Fortran supplies the **ieee_get_rounding_mode()** and **ieee_set_rounding_mode()** procedures in the **ieee_arithmetic** module. These portable procedures retrieve and set the current floating-point rounding mode, respectively.

For example:

```

program fptest
    use, intrinsic :: ieee_arithmetic
    type(ieee_round_type) current_mode
    call ieee_get_rounding_mode( current_mode )
    if ( current_mode == ieee_nearest ) then
        print *, 'Before test: Rounding mode is towards nearest'
        print *, '          2.0 / 3.0 = ', 2.0 / 3.0
        print *, '          -2.0 / 3.0 = ', -2.0 / 3.0
    end if

    call ieee_set_rounding_mode( ieee_up )
    r = 2.0 / 3.0
    print *, 'Round towards +infinity: 2.0 / 3.0 = ', r

    call ieee_set_rounding_mode( ieee_down )
    r = -2.0 / 3.0
    print *, 'Round towards -infinity: -2.0 / 3.0 = ', r
end

```

Notes:

1. Extended-precision floating-point values must only be used in round-to-nearest mode.

2. For thread-safety and reentrancy, the include file `/opt/ibmcmp/xlf/9.1/include/fpdc.h` contains a **THREADLOCAL** directive that is protected by the trigger constant **IBMT**. The invocation commands `xlf_r`, `xlf90_r`, and `xlf95_r` turn on the **-qthreaded** compiler option by default, which in turn implies the trigger constant **IBMT**. If you are including the file `/opt/ibmcmp/xlf/9.1/include/fpdc.h` in code that is not intended to be thread-safe, do not specify **IBMT** as a trigger constant.

Minimizing Rounding Errors

There are several strategies for handling rounding errors and other unexpected, slight differences in calculated results. You may want to consider one or more of the following strategies:

- Minimizing the amount of overall rounding
- Delaying as much rounding as possible to run time
- Ensuring that if some rounding is performed in a mode other than round-to-nearest, *all* rounding is performed in the same mode

Minimizing Overall Rounding

Rounding operations, especially in loops, reduce code performance and may have a negative effect on the precision of computations. Consider using double-precision variables instead of single-precision variables when you store the temporary results of double-precision calculations, and delay rounding operations until the final result is computed.

Delaying Rounding until Run Time

The compiler evaluates floating-point expressions during compilation when it can, so that the resulting program does not run more slowly due to unnecessary run-time calculations. However, the results of the compiler's evaluation might not match exactly the results of the run-time calculation. To delay these calculations until run time, specify the **nofold** suboption of the **-qfloat** option.

The results may still not be identical; for example, calculations in **DATA** and **PARAMETER** statements are still performed at compile time.

The differences in results due to **fold** or **nofold** are greatest for programs that perform extended-precision calculations or are compiled with the **-O** option or both.

Ensuring that the Rounding Mode is Consistent

You can change the rounding mode from its default setting of round-to-nearest. (See for examples.) If you do so, you must be careful that *all* rounding operations for the program use the same mode:

- Specify the equivalent setting on the **-qieee** option, so that any compile-time calculations use the same rounding mode.
- Specify the **rrm** suboption of the **-qfloat** option, so that the compiler does not perform any optimizations that require round-to-nearest rounding mode to work correctly.

For example, you might compile a program like the one in “Selecting the Rounding Mode” on page 254 with this command if the program consistently uses round-to-plus-infinity mode:

```
xlf95 -qieee=plus -qfloat=rrm changes_rounding_mode.f
```

Duplicating the Floating-Point Results of Other Systems

To duplicate the double-precision results of programs on systems with different floating-point architectures (without multiply-add instructions), specify the **nomaf** suboption of the **-qfloat** option. This suboption prevents the compiler from generating any multiply-add operations. This results in decreased accuracy and performance but provides strict conformance to the IEEE standard for double-precision arithmetic.

To duplicate the results of programs where the default size of **REAL** items is different from that on systems running XL Fortran, use the **-qrealize** option (page 187) to change the default **REAL** size when compiling with XL Fortran.

If the system whose results you want to duplicate preserves full double precision for default real constants that are assigned to **DOUBLE PRECISION** variables, use the **-qdp** or **-qrealize** option.

If results consistent with other systems are important to you, include **norsqrt** and **nofold** in the settings for the **-qfloat** option. If you specify the option **-O3**, include **-qstrict** too.

Maximizing Floating-Point Performance

If performance is your primary concern and you want your program to be relatively safe but do not mind if results are slightly different (generally more precise) from what they would be otherwise, optimize the program with the **-O** option, and specify **-qfloat=rsqrt:fltint**. The following section describes the functions of these suboptions:

- The **rsqrt** suboption replaces division by a square root with multiplication by the reciprocal of the root, a faster operation that may not produce precisely the same result.
- The **fltint** suboption speeds up float-to-integer conversions by reducing error checking for overflows. You should make sure that any floats that are converted to integers are not outside the range of the corresponding integer types.

Detecting and Trapping Floating-Point Exceptions

As stated earlier, the IEEE standard for floating-point arithmetic defines a number of exception (or error) conditions that might require special care to avoid or recover from. The following sections are intended to help you make your programs work safely in the presence of such exception conditions while sacrificing the minimum amount of performance.

The floating-point hardware always detects a number of floating-point exception conditions (which the IEEE standard rigorously defines): overflow, underflow, zerodivide, invalid, and inexact.

By default, the only action that occurs is that a status flag is set. The program continues without a problem (although the results from that point on may not be what you expect). If you want to know when an exception occurs, you can arrange for one or more of these exception conditions to generate a signal.

The signal causes a branch to a handler routine. The handler receives information about the type of signal and the state of the program when the signal occurred. It

can produce a core dump, display a listing showing where the exception occurred, modify the results of the calculation, or carry out some other processing that you specify.

The XL Fortran compiler uses the operating system facilities for working with floating-point exception conditions. These facilities indicate the presence of floating-point exceptions by generating **SIGFPE** signals.

Compiler Features for Trapping Floating-Point Exceptions

To turn on XL Fortran exception trapping, compile the program with the **-qfltrap** option and some combination of suboptions that includes **enable**. This option uses trap operations to detect floating-point exceptions and generates **SIGFPE** signals when exceptions occur.

-qfltrap also has suboptions that correspond to the names of the exception conditions. For example, if you are only concerned with handling overflow and underflow exceptions, you could specify something similar to the following:

```
xlf95 -qfltrap=overflow:underflow:enable compute_pi.f
```

You only need **enable** when you are compiling the main program. However, it is very important and does not cause any problems if you specify it for other files, so always include it when you use **-qfltrap**.

An advantage of this approach is that performance impact is relatively low. To further reduce performance impact, you can include the **imprecise** suboption of the **-qfltrap** option. This suboption delays any trapping until the program reaches the start or end of a subprogram.

The disadvantages of this approach include the following:

- It only traps exceptions that occur in code that you compiled with **-qfltrap**, which does not include system library routines.
- It is generally not possible for a handler to substitute results for failed calculations if you use the **imprecise** suboption of **-qfltrap**.

Notes:

1. If your program depends on floating-point exceptions occurring for particular operations, also specify **-qfloat** suboptions that include **nofold**. Otherwise, the compiler might replace an exception-producing calculation with a constant NaN or infinity value, or it might eliminate an overflow in a single-precision operation.
2. The suboptions of the **-qfltrap** option replace an earlier technique that required you to modify your code with calls to the **fpsets** and **fpgets** procedures. You no longer require these calls for exception handling if you use the appropriate **-qfltrap** settings.

Attention: If your code contains **fpsets** calls that enable checking for floating-point exceptions and you do not use the **-qfltrap** option when compiling the whole program, the program will produce unexpected results if exceptions occur, as explained in Table 14 on page 252.

Installing an Exception Handler

When a program that uses the XL Fortran or Linux exception-detection facilities encounters an exception condition, it generates a signal. This causes a branch to whatever handler is specified by the program.

By default, the program stops after producing a core file, which you can use with a debugger to locate the problem. If you want to install a **SIGTRAP** or **SIGFPE** signal handler, use the **-qsigtrap** option. It allows you to specify an XL Fortran handler that produces a traceback or to specify a handler you have written:

```
xlf95 -qfltttrap=ov:und:en pi.f # Dump core on an exception
xlf95 -qfltttrap=ov:und:en -qsigtrap pi.f # Uses the xl_trce handler
xlf95 -qfltttrap=ov:und:en -qsigtrap=return_22_over_7 pi.f # Uses any other handler
```

You can also install an alternative exception handler, either one supplied by XL Fortran or one you have written yourself, by calling the **SIGNAL** subroutine (defined in `/opt/ibmcmp/xlf/9.1/include/fexcp.h`):

```
INCLUDE 'fexcp.h'
CALL SIGNAL(SIGTRAP,handler_name)
CALL SIGNAL(SIGFPE,handler_name)
```

The XL Fortran exception handlers and related routines are:

xl_ieee	Produces a traceback and an explanation of the signal and continues execution by supplying the default IEEE result for the failed computation. This handler allows the program to produce the same results as if exception detection was not turned on.
xl_trce	Produces a traceback and stops the program.
xl_trcedump	Produces a traceback and a core file and stops the program.
xl_sigdump	Provides a traceback that starts from the point at which it is called and provides information about the signal. You can only call it from inside a user-written signal handler. It does not stop the program. To successfully continue, the signal handler must perform some cleanup after calling this subprogram.
xl_trbk	Provides a traceback that starts from the point at which it is called. You call it as a subroutine from your code, rather than specifying it with the -qsigtrap option. It requires no parameters. It does not stop the program.

All of these handler names contain double underscores to avoid duplicating names that you declared in your program. All of these routines work for both **SIGTRAP** and **SIGFPE** signals.

You can use the **-g** compiler option to get line numbers in the traceback listings. The file `/opt/ibmcmp/xlf/9.1/include/fsignal.h` defines a Fortran derived type similar to the sigcontext structure in the `signal.h` system header. You can write a Fortran signal handler that accesses this derived type.

Related Information: “Sample Programs for Exception Handling” on page 262 lists some sample programs that illustrate how to use these signal handlers or write your own. For more information, see **SIGNAL**, in the *Intrinsic Procedures* section in the *XL Fortran Language Reference*.

Controlling the Floating-Point Status and Control Register

Before the `-qfltrap` suboptions or the `-qsigtrap` options, most of the processing for floating-point exceptions required you to change your source files to turn on exception trapping or install a signal handler. Although you can still do so, for any new applications, we recommend that you use the options instead.

To control exception handling at run time, compile without the `enable` suboption of the `-qfltrap` option:

```
xlf95 -qfltrap compute_pi.f      # Check all exceptions, but do not trap.
xlf95 -qfltrap=ov compute_pi.f  # Check one type, but do not trap.
```

Then, inside your program, manipulate the `fpstats` array (defined in the include file `/opt/ibmcomp/xlf/9.1/include/fpdc.h`) and call the `fpsets` subroutine to specify which exceptions should generate traps.

See the sample program that uses `fpsets` and `fpgets` in “Selecting the Rounding Mode” on page 254.

Another method is to use the `set_fpscr_flags()` subroutine in the `xlf_fp_util` module. This subroutine allows you to set the floating-point status and control register flags you specify in the `MASK` argument. Flags that you do not specify in `MASK` remain unaffected. `MASK` must be of type `INTEGER(FPSCR_KIND)`. For example:

```
USE, INTRINSIC :: xlf_fp_util
INTEGER(FPSCR_KIND) SAVED_FPSCR
INTEGER(FP_MODE_KIND) FP_MODE

SAVED_FPSCR = get_fpscr()           ! Saves the current value of
                                   ! the fpscr register.

CALL set_fpscr_flags(TRP_DIV_BY_ZERO) ! Enables trapping of
! ...                               ! divide-by-zero.
SAVED_FPSCR=set_fpscr(SAVED_FPSCR)  ! Restores fpscr register.
```

Another method is to use the `ieee_set_halting_mode` subroutine in the `ieee_exceptions` module. This portable, elemental subroutine allows you to set the halting (trapping) status for any `FPSCR` exception flags. For example:

```
USE, INTRINSIC :: ieee_exceptions
TYPE(IEEE_STATUS_TYPE) SAVED_FPSCR
CALL ieee_get_status(SAVED_FPSCR)  ! Saves the current value of the
                                   ! fpscr register

CALL ieee_set_halting_mode(IEEE_DIVIDE_BY_ZERO, .TRUE.) ! Enabled trapping
! ...                                                    ! of divide-by-zero.

CALL IEEE_SET_STATUS(SAVED_FPSCR) ! Restore fpscr register
```

xlf_fp_util Procedures

The `xlf_fp_util` procedures allow you to query and control the floating-point status and control register (fpscr) of the processor directly. These procedures are more efficient than the `fpsets` and `fpgets` subroutines because they are mapped into inlined machine instructions that manipulate the floating-point status and control register directly.

The intrinsic module, `xlf_fp_util`, contains the interfaces and data type definitions for these procedures and the definitions for the named constants that are needed by the procedures. This module enables type checking of these procedures at

compile time rather than link time. The following files are supplied for the `xlf_fp_util` module:

File names	File type	Locations
xlf_fp_util.mod	module symbol file (32-bit)	/opt/ibmcmp/xlf/9.1/include
	module symbol file (64-bit)	/opt/ibmcmp/xlf/9.1/include64

To use the procedures, you must add a `USE XLF_FP_UTIL` statement to your source file. For more information, see `USE`, in the *XL Fortran Language Reference*.

When compiling with the `-U` option, you must code the names of these procedures in all lowercase.

For a list of the `xlf_fp_util` procedures, see the *Service and Utility Procedures* section in the *XL Fortran Language Reference*.

fpgets and fpsets Subroutines

The `fpsets` and `fpgets` subroutines provide a way to manipulate or query the floating-point status and control register. Instead of calling the operating system routines directly, you pass information back and forth in `fpstat`, an array of logicals. The following table shows the most commonly used array elements that deal with exceptions:

Table 16. Exception Bits to Use with `fpsets` and `fpgets`

Array Element to Set to Enable	Array Element to Check if Exception Occurred	Exception Indicated When <code>.TRUE.</code>
n/a	fpstat(fpfx)	Floating-point exception summary
n/a	fpstat(fpfex)	Floating-point enabled exception summary
fpstat(fpve)	fpstat(fpvx)	Floating-point invalid operation exception summary
fpstat(fpoe)	fpstat(fpox)	Floating-point overflow exception
fpstat(fpue)	fpstat(fpux)	Floating-point underflow exception
fpstat(fpze)	fpstat(fpzx)	Zero-divide exception
fpstat(fpxe)	fpstat(fpxx)	Inexact exception
fpstat(fpve)	fpstat(fpvxsnan)	Floating-point invalid operation exception (signaling NaN)
fpstat(fpve)	fpstat(fpvxisi)	Floating-point invalid operation exception (INF-INF)
fpstat(fpve)	fpstat(fpvxidi)	Floating-point invalid operation exception (INF/INF)
fpstat(fpve)	fpstat(fpvxzdz)	Floating-point invalid operation exception (0/0)
fpstat(fpve)	fpstat(fpvximz)	Floating-point invalid operation exception (INF*0)
fpstat(fpve)	fpstat(fpvxvc)	Floating-point invalid operation exception (invalid compare)

Table 16. Exception Bits to Use with `fpsets` and `fpgets` (continued)

Array Element to Set to Enable	Array Element to Check if Exception Occurred	Exception Indicated When <code>.TRUE.</code>
n/a	<code>fpstat(fpvxsoft)</code>	Floating-point invalid operation exception (software request), PowerPC only
n/a	<code>fpstat(fpvxsqrt)</code>	Floating-point invalid operation exception (invalid square root), PowerPC only
n/a	<code>fpstat(fpvxcvi)</code>	Floating-point invalid operation exception (invalid integer convert), PowerPC only

To explicitly check for specific exceptions at particular points in a program, use `fpgets` and then test whether the elements in `fpstat` have changed. Once an exception has occurred, the corresponding exception bit (second column in the preceding table) is set until it is explicitly reset, except for `fpstat(fpfx)`, `fpstat(fpvx)`, and `fpstat(fpfex)`, which are reset only when the specific exception bits are reset.

An advantage of using the `fpgets` and `fpsets` subroutines (as opposed to controlling everything with suboptions of the `-qfltrap` option) includes control over granularity of exception checking. For example, you might only want to test if an exception occurred anywhere in the program when the program ends.

The disadvantages of this approach include the following:

- You have to change your source code.
- These routines differ from what you may be accustomed to on other platforms.

For example, to trap floating-point overflow exceptions but only in a certain section of the program, you would set `fpstat(fpoe)` to `.TRUE.` and call `fpsets`. After the exception occurs, the corresponding exception bit, `fpstat(fpox)`, is `.TRUE.` until the program runs:

```
call fpgets(fpstat)
fpstat(fpox) = .FALSE.
call fpsets(fpstat) ! resetting fpstat(fpox) to .FALSE.
```

Sample Programs for Exception Handling

`/opt/ibmcomp/xlf/9.1/samples/floating_point` contains a number of sample programs to illustrate different aspects of exception handling:

`fltrap_handler.c` and `fltrap_test.f`

A sample exception handler that is written in C and a Fortran program that uses it.

`xl_ieee.F` and `xl_ieee.c`

Exception handlers that are written in Fortran and C that show how to substitute particular values for operations that produce exceptions. Even when you use support code such as this, the implementation of XL Fortran exception handling does not fully support the exception-handling environment that is suggested by the IEEE floating-point standard.

`check_fpscr.f` and `postmortem.f`

Show how to work with the `fpsets` and `fpgets` procedures and the `fpstats` array.

<code>fhandler.F</code>	Shows a sample Fortran signal handler and demonstrates the <code>xl__sigdump</code> procedure.
<code>xl__trbk_test.f</code>	Shows how to use the <code>xl__trbk</code> procedure to generate a traceback listing without stopping the program.

The sample programs are strictly for illustrative purposes only.

Causing Exceptions for Particular Variables

To mark a variable as “do not use”, you can encode a special value called a signaling NaN in it. This causes an invalid exception condition any time that variable is used in a calculation.

If you use this technique, use the **nans** suboption of the **-qfloat** option, so that the program properly detects all cases where a signaling NaN is used, and one of the methods already described to generate corresponding **SIGFPE** signals.

Notes:

1. Because a signaling NaN is never generated as the result of a calculation and must be explicitly introduced to your program as a constant or in input data, you should not need to use this technique unless you deliberately use signaling NaN values in it.

Minimizing the Performance Impact of Floating-Point Exception Trapping

If you need to deal with floating-point exception conditions but are concerned that doing so will make your program too slow, here are some techniques that can help minimize the performance impact:

- Consider using only a subset of the **overflow**, **underflow**, **zerodivide**, **invalid**, and **inexact** suboptions with the **-qflttrap** option if you can identify some conditions that will never happen or you do not care about. In particular, because an **inexact** exception occurs for each rounding error, you probably should not check for it if performance is important.
- Include the **imprecise** suboption with the **-qflttrap** option, so that your compiler command looks similar to this:

```
xl f90 -qflttrap=underflow:enable:imprecise does_underflows.f
```

imprecise makes the program check for the specified exceptions only on entry and exit to subprograms that perform floating-point calculations. This means that XL Fortran will eventually detect any exception, but you will know only the general area where it occurred, not the exact location.

When you specify **-qflttrap** without **imprecise**, a check for exceptions follows each floating-point operation. If all your exceptions occur during calls to routines that are not compiled with **-qflttrap** (such as library routines), using **imprecise** is generally a good idea, because identifying the exact location will be difficult anyway.

Note that **enable** has no effect if using the **nanq** suboption. **nanq** generates trapping code after each floating point arithmetic, load instruction and procedure returning floating point values even if **imprecise** is specified.

Optimizing XL Fortran Programs

This section provides background information on optimization, guidance on using XL Fortran's optimization features, and details of some XL Fortran optimization techniques.

Simple compilation is the translation or transformation of the source code into an executable or shared object. An optimizing transformation is one that gives your application better overall performance at run time. XL Fortran provides a portfolio of optimizing transformations tailored to the IBM hardware. These transformations can:

- Reduce the number of instructions executed for critical operations.
- Restructure the generated object code to make optimal use of the PowerPC architecture.
- Improve the usage of the memory subsystem.
- Exploit the ability of the architecture to handle large amounts of shared memory parallelization.

Significant performance improvements are possible with relatively little development effort because the compiler is capable of widely applicable and sophisticated program analysis and transformation. Moreover, the compilers enable programming models such as OpenMP, which allow you to write high-performance code.

Optimizations are intended for later phases of application development cycles, such as product release builds. If possible, you should test and debug your code without optimization before attempting to optimize it.

Optimization is controlled by compiler options and directives. However, compiler-friendly programming idioms can be as useful to performance as any of the options or directives. It is no longer necessary nor is it recommended to excessively hand-optimize your code (for example, manually unrolling loops). Unusual constructs can confuse the compiler (and other programmers), and make your application difficult to optimize for new machines. The section *Compiler-Friendly Programming* contains some suggested idioms and programming tips for writing good optimizable code.

It should be noted that not all optimizations are beneficial for all applications. A trade-off usually has to be made between an increase in compile time accompanied by reduced debugging capability and the degree of optimization done by the compiler.

The Philosophy of XL Fortran Optimizations

XL Fortran optimizations can be characterized according to their *aggressiveness*, which determines how much risk they carry. Only the very highest optimization levels perform aggressive optimizations, and even then the risk is limited to slightly different results in a small subset of possible programs.

The less-aggressive optimizations are intended to produce exactly the same results as an equivalent unoptimized program:

- Code that might cause an exception is not moved unless the exception is certain to occur anyway. In the following example, the program could evaluate the expression N/K before the loop because the result is the same for each iteration of the loop:

```

      DO 10 J=1,N
      ...
      IF (K .NE. 0) M(J)=N/K
      ...
10    END

```

However, it is not moved because K might be 0, and computing N/K results in an exception where none occurs in the unoptimized program.

- The rules for IEEE arithmetic are followed more closely than otherwise.³ For example, $X+0.0$ is not folded to X , because IEEE rules require that $-0.0+0.0$ be 0, making $X+0$ equal to $-X$ in this one case.
- Floating-point calculations are not considered associative. For example, XL Fortran evaluates $X*Y*Z$ left-to-right, even though the program might already have computed $Y*Z$, because the results might not be identical.

As the optimization level increases, these restrictions are relaxed where there is an opportunity for a performance improvement:

- Calculations like N/K in the previous example and floating-point operations may be moved or rescheduled because they are unlikely to cause exceptions.
- IEEE conformance is not enforced for rules that are unlikely to be needed. The sign of zero might not be correctly preserved, as in the preceding example. However, this might only be a problem in an extreme case, such as multiplying the wrongly signed zero by infinity and ending up with an infinity of the wrong sign. Floating-point operations that might cause an exception may be moved, rescheduled, or processed so they do not produce an exception.
- Floating-point expressions might be reassociated, so that results might not be identical.

When you specify the highest levels of optimization, XL Fortran assumes that you are requesting speed even at the possibility of some risk, as already explained. If you want as much optimization as possible without the resulting risk, you must add the **-qstrict** compiler option.

The early XL family of compilers adopted a conservative approach to optimization. This was intended to make an optimized program work exactly the same as an unoptimized one, even in extreme cases unlikely to occur in real life. For example, the array reference $A(N)$ might not be optimized, because N might be a huge number so that the program causes a segmentation violation when the address is referenced, and this behavior would be “preserved”. With the industry in general favoring a less conservative approach, XL Fortran’s highest optimization levels now emphasize performance over identical execution between optimized and unoptimized programs.

The different levels of the **-O** option incorporate various optimization techniques that are expected to improve performance for many different kinds of programs. The specialized optimization options, such as **-qipa** and **-qhot**, can improve performance in some kinds of programs but degrade it in others. Therefore, they may require experimentation to determine whether they are appropriate for any given program.

³ If IEEE compliance is a concern for you, you should also specify either the **-qfloat** or **-qstrict** option.

Summary of Compiler Options for Optimization

The following table describes the compiler options that have the most effect on performance. You can experiment with different combinations of compiler options to see which options are most suitable for your application.

Table 17. Compiler options for optimization

Option	Overview
-O0 or -qnoopt	The compiler performs very limited optimization. This is the default optimization level. Before you start optimizing your application, ensure that it compiles and executes successfully at optimization level 0.
-qarch	The compiler generates instructions for the specified processor architecture. This option allows the compiler to take advantage of processor instructions that exist on the specified architecture but do not exist on other architectures.
-qtune	The compiler optimizes the application for the specified processor architecture. The application will run on all of the processors specified by the -qarch option, but its performance will be tuned for the processors specified by the -qtune option.
-O2	The compiler performs basic optimization. This option provides a balance between compilation speed and run-time performance.
-O3	The compiler performs aggressive optimization, including optimizations that are memory-intensive, compile-time-intensive, or both.
-qhot	The compiler performs additional loop optimization, automatic vectorization, and optionally performs array padding. This option is most useful for scientific applications that contain numerical processing. If you want better performance than -O3 with less compile time than -O4 , try -O3 -qhot .
-qcache	The compiler assumes that the processor will have the specified cache configuration. This option can improve performance if all of the processors that execute the application have the same non-default cache configuration.
-qipa	The compiler performs interprocedural analysis to optimize the entire application as a unit. This option is most useful for applications that contain a large number of frequently used routines. In many cases, this option significantly increases compilation time.
-O4	This is equivalent to: -O3 -qipa -qhot -qarch=auto -qtune=auto -qcache=auto
-O5	This is equivalent to: -O4 -qipa=level=2
-qpdf1 -qpdf2	The compiler uses profile-directed feedback to optimize the application based on an analysis of how often your application executes different sections of code.

Choosing an Optimization Level

Optimization requires additional compilation time, but usually results in a faster run time. XL Fortran allows you to select whether you want optimization to be performed at compile time. By default, the compiler performs very limited optimization (**-O0** or **-qnoopt**).

To enable compiler optimization, specify the **-O** compiler option with an optional digit that signifies the level. The following table summarizes compiler behavior at each optimization level.

Optimization levels

Option	Behavior
-qnoopt/-O0	Fast compilation, debuggable code, conserved program semantics.
-O2	Comprehensive low-level optimization; partial debugging support. (This is the same as specifying -O .)
-O3	More extensive optimization; some precision trade-offs.
-O4 and -O5	Interprocedural optimization; loop optimization; automatic machine tuning.

Optimization Level -O2

At optimization level **-O2** (same as **-O**), the compiler performs comprehensive low-level optimization, which includes the following techniques:

- Global assignment of user variables to registers, also known as *graph coloring register allocation*.
- Strength reduction and effective use of addressing modes.
- Elimination of redundant instructions, also known as *common subexpression elimination*
- Elimination of instructions whose results are unused or that cannot be reached by a specified control flow, also known as *dead code elimination*.
- Value numbering (algebraic simplification).
- Movement of invariant code out of loops.
- Compile-time evaluation of constant expressions, also known as *constant propagation*.
- Control flow simplification.
- Instruction scheduling (reordering) for the target machine.
- Loop unrolling and software pipelining.

Minimal debugging information at optimization level **-O2** consists of the following behaviors:

- Externals and parameter registers are visible at procedure boundaries, which are the entrance and exit to a procedure. You can look at them if you set a breakpoint at the entry to a procedure. However, function inlining with **-Q** can eliminate these boundaries and this visibility. This can also happen when the compiler inlines very small functions.
- The **SNAPSHOT** directive creates additional program points for storage visibility by flushing registers to memory. This allows you to view and modify the values of any local or global variable, or of any parameter in your program. You can set a breakpoint at the **SNAPSHOT** and look at that particular area of storage in a debugger.
- The **-qkeepparm** option forces parameters to memory on entry to a procedure so that they can be visible in a stack trace.

Optimization Level -O3

At optimization level **-O3**, the compiler performs more extensive optimization than at **-O2**. The optimizations may be broadened or deepened in the following ways:

- Deeper inner loop unrolling.
- Better loop scheduling.
- Increased optimization scope, typically to encompass a whole procedure.
- Specialized optimizations (those that might not help all programs).
- Optimizations that require large amounts of compile time or space.
- Implicit memory usage limits are eliminated (equivalent to compiling with **-qmaxmem=-1**).

- Implies **-qnostrict**, which allows some reordering of floating-point computations and potential exceptions.

Due to the implicit setting of **-qnostrict**, some precision trade-offs are made by the compiler, such as the following:

- Reordering of floating-point computations.
- Reordering or elimination of possible exceptions (for example, division by zero, overflow).

-O3 optimizations may:

- Require more machine resources during compilation
- Take longer to compile
- Change the semantics of the program slightly

Use the **-O3** option where run-time performance is a crucial factor and machine resources can accommodate the extra compile-time work.

The exact optimizations that are performed depend on a number of factors:

- Whether the program can be rearranged and still execute correctly
- The relative benefit of each optimization
- The machine architecture

Getting the Most out of **-O2** and **-O3**

Here is a recommended approach to using optimization levels **-O2** and **-O3**

- If possible, test and debug your code without optimization before using **-O2**.
- Ensure that your code complies with its language standard. Optimizers assume and rely on that fact that code is standard conformant. Code that is even subtly non-conformant can cause an optimizer to perform incorrect code transformations.

Ensure that subroutine parameters comply with aliasing rules.

- Mark all code that accesses or manipulates data objects by independent input/output processes and independent, asynchronously interrupting processes as **VOLATILE**. For example, mark code which accesses shared variables and pointers to shared variables.
- Compile as much of your code as possible with **-O2**.
- If you encounter problems with **-O2**, check the code for any nonstandard use of aliasing rules before using the **-qalias=nostd** option.
- Next, use **-O3** on as much code as possible.
- If you encounter problems or performance degradations, consider using **-qstrict** or **-qcompact** along with **-O3** where necessary.
- If you still have problems with **-O3**, switch to **-O2** for a subset of files, but consider using **-qmaxmem=-1** or **-qnostrict**, or both.
- If your code works correctly with **-O2** or **-O3**, consider additionally specifying **-qhot** which can perform additional optimizations, especially on loops. Trying **-O3 -qhot** prior to trying **-O4** or **-O5** can improve your code without the compile-time overhead **-qipa** can have that is implied when using **-O4** and **-O5**.

The **-O4** and **-O5** Options

Optimization levels **-O4** and **-O5** automatically activate several other optimization options. Optimization level **-O4** includes:

- Everything from **-O3**
- **-qhot**
- **-qipa**
- **-qarch=auto**
- **-qtune=auto**

- `-qcache=auto`

Optimization level `-O5` includes:

- Everything from `-O4`
- `-qipa=level=2`

If `-O5` is specified on the compile step, then it should be specified on the link step, as well. Although the `-qipa` option is not strictly another optimization level, it extends the optimizations across procedures (even if the procedures are in different files). It enhances the effectiveness of the optimizations that are done by other optimization options, particularly `-O` (at any level). Because it can also increase compile time substantially, you may want to use it primarily for tuning applications that are already debugged and ready to be used. If your application contains a mixture of Fortran and C or C++ code compiled with IBM XL C/C+ compilers, you can achieve additional optimization by compiling and linking all your code with the `-O5` option.

Optimizing for a Target Machine or Class of Machines

Target machine options are options that instruct the compiler to generate code for optimal execution on a given processor or architecture family. By default, the compiler generates code that runs on all supported systems, but perhaps suboptimally on a given system. By selecting appropriate target machine options, you can optimize your application to suit the broadest possible selection of target processors, a range of processors within a given family, or a specific processor. The following compiler options control optimizations affecting individual aspects of the target machine.

Target machine options

Option	Behavior
<code>-q32</code>	Generates code for a 32-bit addressing model (32-bit execution mode).
<code>-q64</code>	Generates code for a 64-bit addressing model (64-bit execution mode).
<code>-qarch</code>	Selects a family of processor architectures, or a specific architecture, for which instruction code should be generated.
<code>-qtune</code>	Biases optimization toward execution on a given processor, without implying anything about the instruction set architecture to use as a target.
<code>-qcache</code>	Defines a specific cache or memory geometry. The defaults are set through <code>-qtune</code> .

Selecting a predefined optimization level sets default values for these individual options.

Related Information: See “`-qarch` Option” on page 100, “`-qtune` Option” on page 216, and “`-qcache` Option” on page 109.

Getting the Most out of Target Machine Options

Try to specify with `-qarch` the smallest family of machines possible that will be expected to run your code reasonably well.

- **-qarch=auto** generates code that may take advantage of instructions available only on the compiling machine (or similar machines).
- The default is **-qarch=ppc**.
- Specifying a **-qarch** option that is not compatible with your hardware, even though your program appears to work, may cause undefined behaviour; the compiler may emit instructions not available on that hardware.

Try to specify with **-qtune** the machine where performance should be best. If you are not sure, let the compiler determine how to best tune for optimization for a given **-qarch** setting.

Before using the **-qcache** option, look at the options sections of the listing using **-qlist** to see if the current settings are satisfactory. The settings appear in the listing itself when the **-qlistopt** option is specified. Modification of cache geometry may be useful in cases where the systems have configurable L2 or L3 cache options or where the execution mode reduces the effective size of a shared level of cache (for example, two-core-per-chip SMP execution on POWER4).

If you decide to use **-qcache**, use **-qhot** or **-qsmp** along with it.

Optimizing Floating-Point Calculations

Special compiler options exist for handling floating-point calculations efficiently. By default, the compiler makes a trade-off to violate certain IEEE floating-point rules in order to improve performance. For example, multiply-add instructions are generated by default because they are faster and produce a more precise result than separate multiply and add instructions. Floating-point exceptions, such as overflow or division by zero, are masked by default. If you need to catch these exceptions, you have the choice of enabling hardware trapping of these exceptions or using software-based checking. The option **-qfltrap** enables software-based checking. On the POWER4, POWER5, or PowerPC 970 processor, hardware trapping is recommended.

Options for handling floating-point calculations

Option	Description
-qfloat	Provides precise control over the handling of floating-point calculations.
-qfltrap	Enables software checking of IEEE floating-point exceptions. This technique is sometimes more efficient than hardware checking because checks can be executed less frequently.

To understand the performance considerations for floating-point calculations with different combinations of compiler options, see “Maximizing Floating-Point Performance” on page 257 and “Minimizing the Performance Impact of Floating-Point Exception Trapping” on page 263.

High-order Transformations (-qhot)

High-order transformations are optimizations that specifically improve the performance of loops and array language. Optimization techniques can include interchange, fusion, and unrolling of loops, and reducing the generation of temporary arrays. The goals of these optimizations include:

- Reducing the costs of memory access through the effective use of caches and translation look-aside buffers.

- Overlapping computation and memory access through effective utilization of the data prefetching capabilities provided by the hardware.
- Improving the utilization of processor resources through reordering and balancing the usage of instructions with complementary resource requirements.
- Taking advantage of VMX instructions to parallelize operations where they are available and the optimizer sees a benefit.

-qhot=vector is the default when **-qhot** is specified. Compiling with **-qhot=vector** transforms some loops to exploit optimized versions of functions rather than the standard versions. The optimized functions reside in a built-in library that includes functions and operations such as reciprocal, square root, and so on. The optimized versions make different trade-offs with respect to precision versus performance. Usage of **-qstrict** implies **-qhot=novector**.

-qhot=simd (short vectorization) optimizes array data to run mathematical operations in parallel where the target architecture allows such operations. Parallel operations occur in 16-byte vector registers. This optimization does not support double-precision floating point mathematics and requires **-qarch=ppc970** to be specified.

Getting the Most out of -qhot

Try using **-qhot** along with **-O3** for all of your code. (The compiler assumes at least **-O2** level for **-qhot**.) It is designed to have a neutral effect when no opportunities for transformation exist.

- If you encounter unacceptably long compile times (this can happen with complex loop nests) or if your performance degrades with the use of **-qhot**, try using **-qhot=novector**, or **-qstrict** or **-qcompact** along with **-qhot**.
- If necessary, deactivate **-qhot** selectively, allowing it to improve some of your code.

Optimizing Loops and Array Language

The **-qhot** option does the following transformations to improve the performance of loops, array language, and memory management:

- Scalar replacement, loop blocking, distribution, fusion, interchange, reversal, skewing, and unrolling
- Reducing generation of temporary arrays

It requires at least level 2 of **-O**. The **-C** option inhibits it.

If you have SMP hardware, you can enable automatic parallelization of loops by specifying the **-qsmp** option. This optimization includes explicitly coded **DO** loops as well as **DO** loops that are generated by the compiler for array language (**WHERE**, **FORALL**, array assignment, and so on). The compiler can only parallelize loops that are independent (each iteration can be computed independently of any other iteration). One case where the compiler will not automatically parallelize loops is where the loops contain I/O, because doing so could lead to unexpected results. In this case, by using the **PARALLEL DO** or work-sharing **DO** directive, you can advise the compiler that such a loop can be safely parallelized. However, the type of I/O must be one of the following:

- Direct-access I/O where each iteration writes to or reads from a different record
- Sequential I/O where each iteration writes to or reads from a different unit
- Stream-access I/O where each iteration uses the **POS=** specifier to write to, or read from, a different part of the file.
- Stream-access I/O where each iteration writes to, or reads from, a different unit.

For more details, refer to the description of the **PARALLEL DO** or work-sharing **DO** directive in the *XL Fortran Language Reference*.

You can use the **-qhot** and **-qsmp** options on:

- Programs with performance bottlenecks that are caused by loops and structured memory accesses
- Programs that contain significant amounts of array language (which can be optimized in the same ways as FORTRAN 77 loops for array operations)

Cost Model for Loop Transformations

The loop transformations performed by the **-qhot** option are controlled by a set of assumptions about the characteristics of typical loops and the costs (in terms of registers used and potential delays introduced) of performing particular transformations.

The cost model takes into consideration:

- The number of available registers and functional units that the processor has
- The configuration of cache memory in the system
- The number of iterations of each loop
- The need to make conservative assumptions to ensure correct results

When the compiler can determine information precisely, such as the number of iterations of a loop, it uses this information to improve the accuracy of the cost model at that location in the program. If it cannot determine the information, the compiler relies on the default assumptions of the cost model. You can change these default assumptions, and thus influence how the compiler optimizes loops, by specifying compiler options:

- **-qassert=nodeps** asserts that none of the loops in the files being compiled have dependencies that extend from one iteration to any other iteration within the same loop. This is known as a loop-carried dependency. If you can assert that the computations performed during iteration n do not require results that are computed during any other iteration, the compiler is better able to rearrange the loops for efficiency.
- **-qassert=itercnt= n** asserts that a “typical” loop in the files that you are compiling will iterate approximately n times. If this is not specified, the assumption is that loops iterate approximately 1024 times. The compiler uses this information to assist in transformations such as putting a high-iteration loop inside a low-iteration one.

It is not crucial to get the value exactly right, and the value does not have to be accurate for every loop in the file. This value is not used if either of the following conditions is true:

- The compiler can determine the exact iteration count.
- You specified the **ASSERT(ITERCNT(n))** directive.

Some of the loop transformations only speed up loops that iterate many times. For programs with many such loops or for programs whose hotspots and bottlenecks are high-iteration loops, specify a large value for n .

A program might contain a variety of loops, some of which are speeded up by these options and others unaffected or even slowed down. Therefore, you might want to determine which loops benefit most from which options, split some loops into different files, and compile the files with the set of options and directives that suits them best.

Unrolling Loops

Loop unrolling involves expanding the loop body to do the work of two, three, or more iterations, and reducing the iteration count proportionately. Benefits to loop unrolling include the following:

- Data dependence delays may be reduced or eliminated
- Loads and stores may be eliminated in successive loop iterations
- Loop overhead may be reduced

Loop unrolling also increases code sizes in the new loop body, which can increase register allocation and possibly cause register spilling. For this reason, unrolling sometimes does not improve performance.

Related Information: See “-qunroll Option” on page 219.

Describing the Hardware Configuration

The **-qtune** setting determines the default assumptions about the number of registers and functional units in the processor. For example, when tuning loops, **-qtune=pwr4** may cause the compiler to unroll most of the inner loops to a depth of two to take advantage of the extra arithmetic units.

The **-qcache** setting determines the blocking factor that the compiler uses when it blocks loops. The more cache memory that is available, the larger the blocking factor.

Efficiency of Different Array Forms

In general, operations on arrays with constant or adjustable bounds, assumed-size arrays, and pointer arrays require less processing than those on automatic, assumed-shape, or deferred-shape arrays and are thus likely to be faster.

Reducing Use of Temporary Arrays

If your program uses array language but never performs array assignments where the array on the left-hand side of the expression overlaps the array on the right-hand side, specifying the option **-qalias=noaryovrlp** can improve performance by reducing the use of temporary array objects.

The **-qhot** option can also eliminate many temporary arrays.

Array Padding

Because of the implementation of the XL Fortran-supported chip target cache architectures, array dimensions that are powers of 2 can lead to decreased cache utilization.

The optional **arraypad** suboption of the **-qhot** option permits the compiler to increase the dimensions of arrays where doing so might improve the efficiency of array-processing loops. If you have large arrays with some dimensions (particularly the first one) that are powers of 2 or if you find that your array-processing programs are slowed down by cache misses or page faults, consider specifying **-qhot=arraypad** or **-qhot=arraypad=*n*** rather than just **-qhot**.

The padding that **-qhot=arraypad** performs is conservative. It also assumes that there are no cases in the source code (such as those created by an **EQUIVALENCE** statement) where storage elements have a relationship that is broken by padding. You can also manually pad array dimensions if you determine that doing so does not affect the program's results.

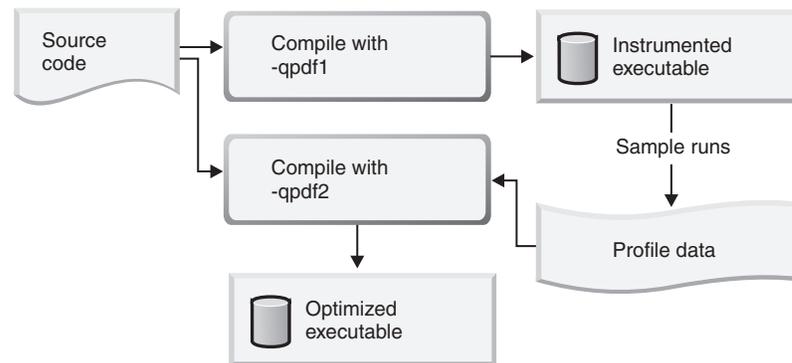
The additional storage taken up by the padding, especially for arrays with many dimensions, might increase the storage overhead of the program to the point where it slows down again or even runs out of storage. For more information, see “-qhot Option” on page 140.

Profile-directed Feedback (PDF)

Profile-directed feedback is a two-stage compilation process that lets you provide the compiler with data characteristic of typical program behavior. An instrumented executable is run in a number of different scenarios for an arbitrary amount of time, producing as a side effect a profile data file. A second compilation using the profile data produces an optimized executable.

PDF should be used mainly on code that has rarely executed conditional error handling or instrumentation. The technique has a neutral effect in the absence of firm profile information, but is not recommended if insufficient or uncharacteristic data is all that is available.

The following diagram illustrates the PDF process.



The two stages of the process are controlled by the compiler options **-qpdf1** and **-qpdf2**. Stage 1 is a regular compilation using an arbitrary set of optimization options and **-qpdf1**, that produces an executable or shared object that can be run in a number of different scenarios for an arbitrary amount of time. Stage 2 is a recompilation using the same options, except **-qpdf2** is used instead of **-qpdf1**, during which the compiler consumes previously collected data for the purpose of path-biased optimization.

Using Profile-directed Feedback (PDF)

You can optimize an application based on an analysis of how often it executes different sections of code, as follows:

1. Compile the application with **-qpdf1**.
2. Run the application using a typical data set or several typical data sets. When the application exits, it writes profiling information in the PDF file.
3. Compile the application with **-qpdf2**.

If you want to see which functions are used the most often, do the following:

1. Compile the application with **-qpdf1 -qshowpdf**.
2. Run the application using a typical data set or several typical data sets. The application writes more detailed profiling information in the PDF file.
3. Use the **showpdf** utility to view the information in the PDF file.

You can take more control of the PDF file generation, as follows:

1. Compile the application with **-qpdf1**.
2. Run the application using a typical data set or several typical data sets. This produces a PDF file in the current directory.
3. Copy the application to another directory and run it again. This produces a PDF file in the second directory.
4. Repeat the previous step as often as you want.
5. Use the **mergepdf** utility to combine the PDF files into one PDF file. For example, if you produce three PDF files that represent usage patterns that will occur 53%, 32%, and 15% of the time respectively, you can use this command:

```
mergepdf -r 53 path1 -r 32 path2 -r 15 path3
```
6. Compile the application with **-qpdf2**.

To erase the information in the PDF directory, use the **cleanpdf** or **resetpdf** utility.

Optimizing Conditional Branching

The **-qpdf** option helps to fine-tune the areas around conditional branches so that the default choices correspond to the most likely execution paths. Sometimes instructions from the more likely execution path run before the branch, in parallel with other instructions so that there is no slowdown.

Because the **-qpdf** option requires some extra compilation overhead and sample execution that uses representative data, you should use it mainly near the end of the development cycle.

Related Information: See “-qpdf Option” on page 175.

Interprocedural Analysis (-qipa)

Interprocedural analysis (IPA) enables the compiler to optimize across different files (whole-program analysis), and can result in significant performance improvements. Interprocedural analysis can be specified on the compile step only, or on both compile and link steps (*whole program* mode). Whole program mode expands the scope of optimization to an entire program unit, which can be an executable or shared object. Whole program IPA analysis can consume significant amounts of memory and time when compiling or linking large programs.

IPA is enabled by the **-qipa** option. A summary of the effects of the most commonly used suboptions follows.

Commonly used **-qipa** suboptions

Suboption	Behavior
level=0	Program partitioning and simple interprocedural optimization, which consists of: <ul style="list-style-type: none">• Automatic recognition of standard libraries.• Localization of statically bound variables and procedures.• Partitioning and layout of procedures according to their calling relationships, which is also referred to as their <i>call affinity</i>. (Procedures that call each other frequently are located closer together in memory.)• Expansion of scope for some optimizations, notably register allocation.

- level=1** Inlining and global data mapping. Specifically,
- Procedure inlining.
 - Partitioning and layout of static data according to reference affinity. (Data that is frequently referenced together will be located closer together in memory.)
- This is the default level when **-qipa** is specified.
- level=2** Global alias analysis, specialization, interprocedural data flow.
- Whole-program alias analysis. This level includes the disambiguation of pointer dereferences and indirect function calls, and the refinement of information about the side effects of a function call.
 - Intensive intraprocedural optimizations. This can take the form of value numbering, code propagation and simplification, code motion into conditions or out of loops, elimination of redundancy.
 - Interprocedural constant propagation, dead code elimination, pointer analysis.
 - Procedure specialization (cloning).
- inline=inline-options**
Provides precise user control of inlining.
- fine_tuning* Other values for **-qipa=** provide the ability to specify the behavior of library code, tune program partitioning, read commands from a file, and so on.

Getting the Most from **-qipa**

It is not necessary to compile everything with **-qipa**, but try to apply it to as much of your program as possible. Here are some suggestions.

- When specifying optimization options in a makefile, remember to use the compiler command (**xl**, **xlF90**, and so on) to link, and to include all compiler options on the link step.
- **-qipa** works when building executables or shared objects, but always compile main and exported functions with **-qipa**.
- When compiling and linking separately, use **-qipa=noobject** on the compile step for faster compilation.
- Ensure that there is enough space in **/tmp** (at least 200 MB), or use the **TMPDIR** environment variable to specify a different directory with sufficient free space.
- The **level** suboption is a throttle. Try varying it if link time is too long. Compiling with **-qipa=level=0** can be very beneficial for little additional link time.
- Look at the generated code after compiling with **-qlist** or **-qipa=list**. If too few or too many functions are inlined, consider using **-qipa=inline** or **-qipa=noinline**. To control inlining of a specific function, use **-Q+** and **-Q-**.
- If your application contains a mixture of Fortran and C or C++ code compiled with IBM XL C/C+ compilers, you can achieve additional optimization by compiling all your code with the **-qipa** option.

Optimizing Subprogram Calls

If a program has many subprogram calls, you can use the **-qipa=inline** option to turn on inlining, which reduces the overhead of such calls. Consider using the **-p** or **-pg** option with **gprof** to determine which subprograms are called most frequently and to list their names on the command line.

To make inlining apply to calls where the calling and called subprograms are in different scopes, include the **-qipa** option.

```
# Let the compiler decide (relatively cautiously) what to inline.
xlf95 -O3 -qipa=inline inline.f

# Encourage the compiler to inline particular subprograms.
xlf95 -O3 -qipa=inline=called_100_times,called_1000_times inline.f

# Explicitly extend the inlining to calls across multiple files.
xlf95 -O3 -qipa=inline=called_100_times,called_1000_times -qipa inline.f
```

Related Information: See “-Q Option” on page 92 and “-qipa Option” on page 149.

Finding the Right Level of Inlining

Getting the right amount of inlining for a particular program may require some work on your part. The compiler has a number of safeguards and limits to avoid doing an excessive amount of inlining. Otherwise, it might perform less overall optimization because of storage constraints during compilation, or the resulting program might be much larger and run slower because of more frequent cache misses and page faults. However, these safeguards may prevent the compiler from inlining subprograms that you do want inlined. If this happens, you will need to do some analysis or rework or both to get the performance benefit.

As a general rule, consider identifying a few subprograms that are called most often, and inline only those subprograms.

Some common conditions that prevent **-qipa=inline** from inlining particular subprograms are:

- The calling and called procedures are in different scopes. If so, you can use the **-qipa** option to enable cross-file inlining.
- After the compiler has expanded a subprogram by a certain amount as a result of inlining, it does not inline subsequent calls from that subprogram. Again, there are different limits, which depend on whether the subprogram being called is named by a **-qipa=inline** option.

Consider an example with three procedures: **A** is the caller, and **B** and **C** are at the upper size limit for automatic inlining. They are all in the same file, which is compiled like this:

```
xlf -qipa=inline=c file.f
```

The **-qipa=inline** means that calls to **C** are more likely to be inlined. If **B** and **C** were twice as large, calls to **B** would not be inlined at all, while some calls to **C** could still be inlined.

Although these limits might prevent some calls from **A** to **B** or **A** to **C** from being inlined, the process starts over after the compiler finishes processing **A**.

- Any interface errors, such as different numbers, sizes, or types of arguments or return values, might prevent a call from being inlined. Make sure that interface blocks for the procedures being called are defined properly.
- Actual or potential aliasing of dummy arguments or automatic variables might prevent a procedure from being inlined. For example, inlining might not occur in the following cases:
 - If you compile the file containing either the calling or called procedure with the option **-qalias=nostd** and there are any arguments to the procedure being called

- If there are more than approximately 31 arguments to the procedure being called
- If any automatic variables in the called procedures are involved in an **EQUIVALENCE** statement
- If the same variable argument is passed more than once in the same call: for example, **CALL SUB(X,Y,X)**
- Some procedures that use computed **GO TO** statements, where any of the corresponding statement labels are also used in an **ASSIGN** statement, might not be inlined.

To change the size limits that control inlining, you can specify **-qipa=limit=*n***, where *n* is 0 through 9. Larger values allow more inlining.

It is possible that C/C++ functions can be inlined into Fortran programs and visa-versa during link-time optimizations. The C/C++ code would have to be compiled using the IBM XL C/C++ compilers with **-qipa** and a compatible option set to that used in the XLF compilation.

Shared-memory Parallelism (-qsmp)

Some IBM processors are capable of shared-memory parallel processing. Compile with **-qsmp** to generate the threaded code needed to exploit this capability. The option implies a **-O2** optimization level. The default behavior for the option without suboptions is to do automatic parallelization with optimization.

The most commonly used **-qsmp** suboptions are summarized in the following table.

Commonly used **-qsmp** suboptions

Suboption	Behavior
auto	Instructs the compiler to automatically generate parallel code where possible without user assistance. This option also recognizes all the SMP directives.
omp	Enforces compliance with the OpenMP Fortran API for specifying explicit parallelism. Note that -qsmp=omp is currently incompatible with -qsmp=auto .
opt	Instructs the compiler to optimize as well as parallelize. The optimization is equivalent to -O2 -qhot in the absence of other optimization options. The default setting of -qsmp is -qsmp=auto:noomp:opt .
<i>suboptions</i>	Other values for the suboption provide control over thread scheduling, nested parallelism, locking, and so on.

Getting the Most out of -qsmp

- Use **-qsmp=omp:noauto** if you are compiling an OpenMP program and do not want automatic parallelization. By default, the parallelization performed is both explicit and automatic.
- Before using **-qsmp** with automatic parallelization, test your programs using optimization and **-qhot** in a single-threaded manner.
- Always use the reentrant compiler invocations (the **_r** command invocations, like **xlf_r**) when using **-qsmp**.
- By default, the runtime uses all available processors. Do not set the **XL SMP_OPTS=PARTHDS** or **OMP_NUM_THREADS** variables unless you want to use fewer than the number of available processors. You might want to set the number of executing threads to a small number or to 1 to ease debugging.
- If you are using a dedicated machine or node, consider setting the **SPINS** and **YIELDS** variables (suboptions of **XL SMP_OPTS**) to 0. Doing so prevents the operating system from intervening in the scheduling of threads across synchronization boundaries such as barriers.

- When debugging an OpenMP program, try using **-qsmp=noopt** (without **-O**) to make the debugging information produced by the compiler more precise. You can also use the **SNAPSHOT** directive to create additional program points for storage visibility by flushing registers to memory.

Other Program Behavior Options

The precision of compiler analyses is significantly affected by instructions that can read or write memory. Aliasing pertains to alternate names for things, which in this context are references to memory. A reference to memory can be direct, as in the case of a named symbol, or indirect, as in the case of a pointer or dummy argument. A function call might also reference memory indirectly. Apparent references to memory that are false, that is, that do not actually reference some location assumed by the compiler, constitute barriers to compiler analysis.

Fortran defines a rule that dummy argument references may not overlap other dummy arguments or externally visible symbols during the execution of a subprogram.

The compiler performs sophisticated analyses, attempting to refine the set of possible aliases for pointer dereferences and calls. However, a limited scope and the absence of values at compile time constrain the effectiveness of these analyses. Increasing the optimization level, in particular, applying interprocedural analysis (that is, compiling with **-qipa**), can contribute to better aliasing.

Programs that violate language aliasing rules, as summarized above, commonly execute correctly without optimization or with low optimization levels, but can begin to fail when higher levels of optimization are attempted. The reason is that more aggressive optimizations take better advantage of aliasing information and can therefore expose subtly incorrect program semantics.

Options related to these issues are **-qstrict** and **-qalias**. Their behaviors are summarized in the table below.

Program behavior options

Option	Description
-qstrict, -qnostrict	Allows the compiler to reorder floating-point calculations and potentially excepting instructions. A potentially excepting instruction is one that may raise an interrupt due to erroneous execution (for example, floating-point overflow, a memory access violation). The default is -qstrict with -qnoopt and -O2 ; -qnostrict with -O3, -O4, and -O5 .
-qalias	Allows the compiler to assume that certain variables do not refer to overlapping storage. The focus is on the overlap of dummy arguments and array assignments in Fortran.

Other Performance Options

Options are provided to control particular aspects of optimization. They are often enabled as a group or given default values when a more general optimization option is enabled.

Selected compiler options for optimizing performance

Option	Description
-qcompact	Chooses reduction of final code size over a reduction in execution time when a choice is necessary. Can be used to constrain -O3 and higher optimizations.
-qsmallstack	Instructs the compiler to limit the use of stack storage in the program. Doing so may increase heap usage.
-qunroll	Independently controls loop unrolling. Is implicitly activated under -O3 and higher optimizations.
-qtbtable	Controls the generation of traceback table information. 64-bit mode only.
-qunwind	Informs the compiler that the stack can be unwound while a routine in this compilation is active. In other words, the compiler is informed that the application may or does rely on program stack unwinding mechanisms.
-qnounwind	Informs the compiler that the stack will not be unwound while any routine in this compilation is active. The -qnounwind option enables optimization prologue tailoring, which reduces the number of saves and restores of nonvolatile registers.

Debugging Optimized Code

Debugging optimized programs presents special problems. Optimization may change the sequence of operations, add or remove code, and perform other transformations that make it difficult to associate the generated code with the original source statements. For example, the optimizer may remove all stores to a variable and keep it alive only in registers. Most debuggers are incapable of following this and it will appear as though that variable is never updated.

First debug your program, then recompile it with any optimization options, and test the optimized program before placing the program into production. If the optimized code does not produce the expected results, isolate the specific optimization problems in another debugging session.

The following table presents options that provide specialized information, which can be helpful during the development of optimized code.

Diagnostic options

Option	Behavior
-qlist	Instructs the compiler to emit an object listing. The object listing includes hex and pseudo-assembly representations of the generated instructions, traceback tables, and text constants.
-qreport	Instructs the compiler to produce a report of the loop transformations it performed and how the program was parallelized. The option is enabled when -qhot or -qsmp is specified.
-qinitauto	Instructs the compiler to emit code that initializes all automatic variables to a given value.
-qipa=list	Instructs the compiler to emit an object listing that provides information for IPA optimization.

Different Results in Optimized Programs

Here are some reasons why an optimized program might produce different results from those of an unoptimized one:

- Optimized code can fail if a program contains code that is not valid. For example, failure can occur if the program passes an actual argument that also appears in a common block in the called procedure, or if two or more dummy arguments are associated with the same actual argument.
- If a program that worked without optimization fails when compiled with it, check the cross-reference listing and the execution flow of the program for variables that are used before they are initialized. Compile with the `-qinitauto=hex_value` option to try to produce the incorrect results consistently. For example, using `-qinitauto=FF` gives **REAL** and **COMPLEX** variables an initial value of "negative not a number" (-NAN). Any operations on these variables will also result in NAN values. Other bit patterns (*hex_value*) may yield different results and provide further clues as to what is going on. (Programs with uninitialized variables may appear to work properly when compiled without optimization, because of the default assumptions the compiler makes, but may fail when compiled with optimization. Similarly, a program may appear to execute correctly when optimized, but fails at lower optimization levels or when run in a different environment.)
- Use with caution debugging techniques that rely on examining values in storage. The compiler might have deleted or moved a common expression evaluation. It might have assigned some variables to registers, so that they do not appear in storage at all.

Related Information: See “-g Option” on page 82, “-qinitauto Option” on page 144, and “Problem Determination and Debugging” on page 325.

Compiler-friendly Programming

Compiler-friendly programming idioms can be as useful to performance as any of the options or directives. Here are some suggestions.

General

- Where possible, use command invocations like `xlf90` or `xlf95` to enhance standards conformance and code portability. If this is not possible, consider using the `-qnosave` option to have all local variables be automatic; doing this provides more opportunities for optimization.
- Use modules to group related subroutines and functions.
- Consider using the highly tuned **MASS** and **ESSL** libraries rather than custom implementations or generic libraries.

Hand-tuning

- Do not excessively hand-optimize your code. Unusual constructs can confuse the compiler (and other programmers), and make your application difficult to optimize for new machines.
- Do limited hand tuning of small functions by inlining.
- Avoid breaking your program into too many small functions as this can increase the percentage of time the program spends in dealing with call overhead. If you choose to use many small functions, seriously consider using `-qipa`.

Variables

- Avoid unnecessary use of global variables and pointers. When using them in a loop, load them into a local variable before the loop and store them back after.
- Use the **INTENT** statement to describe usage of parameters.

Conserving storage

- Use register-sized integers (INTEGER(4) or INTEGER(8) data type) for scalars.
- Use the smallest floating-point precision appropriate to your computation.
- When writing new code, use module variables rather than common blocks for global storage.
- Use the **CONTAINS** statement only to share thread local storage.

Pointers

- Obey all language aliasing rules. Try to avoid using **-qalias=nostd**.
- Limit the use of **ALLOCATABLE** arrays and **POINTER** variables to situations which demand dynamic allocation.

Arrays

- Use local variables wherever possible for loop index variables and bounds.
- Keep array index expressions as simple as possible. Where indexing needs to be indirect, consider using the **PERMUTATION** directive.
- When using array assignment or **WHERE** statements, pay close attention to the generated code with **-qlist** or **-qreport**. If performance is inadequate, consider using **-qhot** or rewriting array language in loop form.

Implementation Details of XL Fortran Input/Output

This section discusses XL Fortran support (through extensions and platform-specific details) for the Linux file system.

Related Information: See “-qposition Option” on page 184 and “Mixed-Language Input and Output” on page 302.

Implementation Details of File Formats

XL Fortran implements files in the following manner:

Sequential-access unformatted files:

An integer that contains the length of the record precedes and follows each record. The length of the integer is 4 bytes for 32-bit applications. It is 4 bytes if you set the **uwidth** run-time option to 32 (the default) and it is 8 bytes if you set **uwidth** to 64 for 64-bit applications.

Sequential-access formatted files:

XL Fortran programs break these files into records while reading, by using each newline character (X'0A') as a record separator.

On output, the input/output system writes a newline character at the end of each record. Programs can also write newline characters for themselves. This practice is not recommended because the effect is that the single record that appears to be written is treated as more than one record when being read or backspaced over.

Direct access files:

XL Fortran simulates direct-access files with files whose length is a multiple of the record length of the XL Fortran file. You must specify, in an **OPEN** statement, the record length (**RECL**) of the direct-access file. XL Fortran uses this record length to distinguish records from each other.

For example, the third record of a direct-access file of record length 100 bytes would start at the 201st byte of the single record of a Linux file and end at the 300th byte.

If the length of the record of a direct-access file is greater than the total amount of data you want to write to the record, XL Fortran pads the record on the right with blanks (X'20').

Stream-access unformatted files:

Unformatted stream files are viewed as a collection of file storage units. In XL Fortran, a file storage unit is one byte.

A file connected for unformatted stream access has the following properties:

- The first file storage unit has position 1. Each subsequent file storage unit has a position that is one greater than that of the preceding one.
- For a file that can be positioned, file storage units need not be read or written in the order of their position. Any file storage unit may be read from the file while it is connected to a unit, provided that the file storage unit has been written since the file was created, and if a **READ** statement for the connection is permitted.

Stream-access formatted files:

A record file connected for formatted stream access has the following properties:

- Some file storage units may represent record markers. The record marker is the newline character (X'0A').
- The file will have a record structure in addition to the stream structure.
- The record structure is inferred from the record markers that are stored in the file.
- Records can have any length up to the internal limit allowed by XL Fortran (See Appendix D, "XL Fortran Internal Limits," on page 361.)
- There may or may not be a record marker at the end of the file. If there is no record marker at the end of the file, the final record is incomplete, but not empty.

A file connected for formatted stream access has the following properties:

- The first file storage unit has position 1. Each subsequent file storage unit has a position that is greater than that of the preceding one. Unlike unformatted stream access, the positions of successive file storage units are not always consecutive.
- The position of a file connected for formatted stream access can be determined by the **POS=** specifier in an **INQUIRE** statement.
- For a file that can be positioned, the file position can be set to a value that was previously identified by the **POS=** specifier in **INQUIRE**.

File Names

You can specify file names as either relative (such as **file**, **dir/file**, or **../file**) or absolute (such as **/file** or **/dir/file**). The maximum length of a file name (the full path name) is 4095 characters, even if you only specify a relative path name in the I/O statement. The maximum length of a file name with no path is 255 characters.

You must specify a valid file name in such places as the following:

- The **FILE=** specifier of the **OPEN** and **INQUIRE** statements
- **INCLUDE** lines

Related Information: To specify a file whose location depends on an environment variable, you can use the **GETENV** intrinsic procedure to retrieve the value of the environment variable:

```
character(100) home, name
call getenv('HOME', value=home)
! Now home = $HOME + blank padding.
! Construct the complete path name and open the file.
name=trim(home) // '/remainder/of/path'
open (unit=10, file=name)
...
end
```

Preconnected and Implicitly Connected Files

Units 0, 5, and 6 are preconnected to standard error, standard input, and standard output, respectively, before the program runs.

All other units can be implicitly connected when an **ENDFILE**, **PRINT**, **READ**, **REWIND**, or **WRITE** statement is performed on a unit that has not been opened. Unit *n* is connected to a file that is named **fort.n**. These files need not exist, and XL Fortran does not create them unless you use their units.

Note: Because unit 0 is preconnected for standard error, you cannot use it for the following statements: **CLOSE**, **ENDFILE**, **BACKSPACE**, **REWIND**, and direct or stream input/output. You can use it in an **OPEN** statement only to change the values of the **BLANK=**, **DELIM=**, or **PAD=** specifiers.

You can also implicitly connect units 5 and 6 (and *) by using I/O statements that follow a **CLOSE**:

```

WRITE (6,10) "This message goes to stdout."
CLOSE (6)
WRITE (6,10) "This message goes in the file fort.6."
PRINT *, "Output to * now also goes in fort.6."
10  FORMAT (A)
END

```

The **FORM=** specifier of implicitly connected files has the value **FORMATTED** before any **READ**, **WRITE**, or **PRINT** statement is performed on the unit. The first such statement on such a file determines the **FORM=** specifier from that point on: **FORMATTED** if the formatting of the statement is format-directed, list-directed, or namelist; and **UNFORMATTED** if the statement is unformatted.

Preconnected files also have **FORM='FORMATTED'**, **STATUS='OLD'**, and **ACTION='READWRITE'** as default specifier values.

The other properties of a preconnected or implicitly connected file are the default specifier values for the **OPEN** statement. These files always use sequential access.

If you want XL Fortran to use your own file instead of the **fort.n** file, you can either specify your file for that unit through an **OPEN** statement or create a symbolic link before running the application. In the following example, there is a symbolic link between **myfile** and **fort.10**:

```
ln -s myfile fort.10
```

When you run an application that uses the preconnected file **fort.10** for input/output, XL Fortran uses the file **myfile** instead. The file **fort.10** exists, but only as a symbolic link. The following command will remove the symbolic link, but will not affect the existence of **myfile**:

```
rm fort.10
```

File Positioning

Table 18. Position of the File Pointer When a File Is Opened with No **POSITION=** Specifier

-qposition suboptions	Implicit OPEN		Explicit OPEN					
			STATUS = 'NEW'		STATUS = 'OLD'		STATUS = 'UNKNOWN'	
	File exists	File does not exist	File exists	File does not exist	File exists	File does not exist	File exists	File does not exist
option not specified	Start	Start	Error	Start	Start	Error	Start	Start
appendold	Start	Start	Error	Start	End	Error	Start	Start
appendunknown	Start	Start	Error	Start	Start	Error	End	Start
appendold and appendunknown	Start	Start	Error	Start	End	Error	End	Start

I/O Redirection

You can use the redirection operator on the command line to redirect input to and output from your XL Fortran program. How you specify and use this operator depends on which shell you are running. Here is a **bash** example:

```
$ cat redirect.f
      write (6,*) 'This goes to standard output'
      write (0,*) 'This goes to standard error'
      read (5,*) i
      print *,i
      end
$ xlf95 redirect.f
** _main      === End of Compilation 1 ===
1501-510  Compilation successful for file redirect.f.
$ # No redirection. Input comes from the terminal. Output goes to
$ # the screen.
$ a.out
  This goes to standard output
  This goes to standard error
4
  4
$ # Create an input file.
$ echo >stdin 2
$ # Redirect each standard I/O stream.
$ a.out >stdout 2>stderr <stdin
$ cat stdout
  This goes to standard output
2
$ cat stderr
  This goes to standard error
```

Refer to your man pages for more information on redirection.

How XLF I/O Interacts with Pipes, Special Files, and Links

You can access regular operating system files and blocked special files by using sequential-access, direct-access, or stream-access methods.

You can only access pseudo-devices, pipes, and character special files by using sequential-access methods, or stream-access without using the **POS=** specifier.

When you link files together, you can use their names interchangeably, as shown in the following example:

```
OPEN (4, FILE="file1")
OPEN (4, FILE="link_to_file1", PAD="NO") ! Modify connection
```

Do not specify the **POSITION=** specifier as **REWIND** or **APPEND** for pipes.

Do not specify **ACTION='READWRITE'** for a pipe.

Do not use the **BACKSPACE** statement on files that are pseudo-devices or character special files.

Do not use the **REWIND** statement on files that are pseudo-devices or pipes.

Default Record Lengths

If a pseudo-device, pipe, or character special file is connected for formatted or unformatted sequential access with no **RECL=** specifier, or for formatted stream access, the default record length is 32 768 rather than 2 147 483 647, which is the default for sequential-access files connected to random-access devices. (See the **default_recl** run-time option.)

In certain cases, the default maximum record length for formatted files is larger, to accommodate programs that write long records to standard output. If a unit is connected to a terminal for formatted sequential access and there is no explicit **RECL=** qualifier in the **OPEN** statement, the program uses a maximum record length of 2 147 483 646 ($2^{31}-2$) bytes, rather than the usual default of 32 768 bytes. When the maximum record length is larger, formatted I/O has one restriction: **WRITE** statements that use the **T** or **TL** edit descriptors must not write more than 32 768 bytes. This is because the unit's internal buffer is flushed each 32 768 bytes, and the **T** or **TL** edit descriptors will not be able to move back past this boundary.

File Permissions

A file must have the appropriate permissions (read, write, or both) for the corresponding operation being performed on it.

When a file is created, the default permissions (if the **umask** setting is 000) are both read and write for user, group, and other. You can turn off individual permission bits by changing the **umask** setting before you run the program.

Selecting Error Messages and Recovery Actions

By default, an XLF-compiled program continues after encountering many kinds of errors, even if the statements have no **ERR=** or **IOSTAT=** specifiers. The program performs some action that might allow it to recover successfully from the bad data or other problem.

To control the behavior of a program that encounters errors, set the **XLFRTEOPTS** environment variable, which is described in "Setting Run-Time Options" on page 35, before running the program:

- To make the program stop when it encounters an error instead of performing a recovery action, include **err_recovery=no** in the **XLFRTEOPTS** setting.
- To make the program stop issuing messages each time it encounters an error, include **xrf_messages=no**.
- To disallow XL Fortran extensions to Fortran 90 at run time, include **langlvl=90std**. To disallow XL Fortran extensions to Fortran 95 at run time, include **langlvl=95std**. To disallow XL Fortran extensions to Fortran 2003 behaviour at run time, include **langlvl=2003std**. These settings, in conjunction with the **-qlanglvl** compiler option, can help you locate extensions when preparing to port a program to another platform.

For example:

```
# Switch defaults for some run-time settings.
XLFRTEOPTS="err_recovery=no:cnverr=no"
export XLFRTEOPTS
```

If you want a program always to work the same way, regardless of environment-variable settings, or want to change the behavior in different parts of the program, you can call the **SETRTEOPTS** procedure:

```

PROGRAM RTEOPTS
USE XLFUTILITY
CALL SETRTEOPTS("err_recovery=no") ! Change setting.
... some I/O statements ...
CALL SETRTEOPTS("err_recovery=yes") ! Change it back.
... some more I/O statements ...
END

```

Because a user can change these settings through the **XLFRTEOPTS** environment variable, be sure to use **SETRTEOPTS** to set all the run-time options that might affect the desired operation of the program.

Flushing I/O Buffers

To protect data from being lost if a program ends unexpectedly, you can use the **FLUSH** statement or the **flush_** subroutine to write any buffered data to a file. (The **FLUSH** statement is recommended for better portability.) The following example shows use of the **flush_** subroutine:

```

USE XLFUTILITY
INTEGER, PARAMETER :: UNIT=10

DO I=1,1000000
  WRITE (10,*) I
  CALL MIGHT_CRASH
! If the program ends in the middle of the loop, some data
! may be lost.
END DO

DO I=1,1000000
  WRITE (10,*) I
  CALL FLUSH_(UNIT)
  CALL MIGHT_CRASH
! If the program ends in the middle of the loop, all data written
! up to that point will be safely in the file.
END DO

END

```

Related Information: See “Mixed-Language Input and Output” on page 302 and the **FLUSH** statement in the *XL Fortran Language Reference*.

Choosing Locations and Names for Input/Output Files

If you need to override the default locations and names for input/output files, you can use the following methods without making any changes to the source code.

Naming Files That Are Connected with No Explicit Name

To give a specific name to a file that would usually have a name of the form **fort.unit**, you must set the run-time option **unit_vars** and then set an environment variable with a name of the form **XLFUNIT_unit** for each scratch file. The association is between a unit number in the Fortran program and a path name in the file system.

For example, suppose that the Fortran program contains the following statements:

```

OPEN (UNIT=1, FORM='FORMATTED', ACCESS='SEQUENTIAL', RECL=1024)
...
OPEN (UNIT=12, FORM='UNFORMATTED', ACCESS='DIRECT', RECL=131072)
...
OPEN (UNIT=123, FORM='UNFORMATTED', ACCESS='SEQUENTIAL', RECL=997)

```

```

XLFRTOPSIS="unit_vars=yes"      # Allow overriding default names.
XLFUNIT_1="/tmp/molecules.dat" # Use this named file.
XLFUNIT_12="./data/scratch"    # Relative to current directory.
XLFUNIT_123="/home/user/data"  # Somewhere besides /tmp.
export XLFRTOPSIS XLFUNIT_1 XLFUNIT_12 XLFUNIT_123

```

Notes:

1. The `XLFUNIT_number` variable name must be in uppercase, and *number* must not have any leading zeros.
2. `unit_vars=yes` might be only part of the value for the `XLFRTOPSIS` variable, depending on what other run-time options you have set. See “Setting Run-Time Options” on page 35 for other options that might be part of the `XLFRTOPSIS` value.
3. If the `unit_vars` run-time option is set to `no` or is undefined or if the applicable `XLFUNIT_number` variable is not set when the program is run, the program uses a default name (`fort.unit`) for the file and puts it in the current directory.

Naming Scratch Files

To place all scratch files in a particular directory, set the `TMPDIR` environment variable to the name of the directory. The program then opens the scratch files in this directory. You might need to do this if your `/tmp` directory is too small to hold the scratch files.

To give a specific name to a scratch file, you must do the following:

1. Set the run-time option `scratch_vars`.
2. Set an environment variable with a name of the form `XLFSRATCH_unit` for each scratch file.

The association is between a unit number in the Fortran program and a path name in the file system. In this case, the `TMPDIR` variable does not affect the location of the scratch file.

For example, suppose that the Fortran program contains the following statements:

```

OPEN (UNIT=1, STATUS='SCRATCH', &
      FORM='FORMATTED', ACCESS='SEQUENTIAL', RECL=1024)
...
OPEN (UNIT=12, STATUS='SCRATCH', &
      FORM='UNFORMATTED', ACCESS='DIRECT', RECL=131072)
...
OPEN (UNIT=123, STATUS='SCRATCH', &
      FORM='UNFORMATTED', ACCESS='SEQUENTIAL', RECL=997)
XLFRTOPSIS="scratch_vars=yes"      # Turn on scratch file naming.
XLFSRATCH_1="/tmp/molecules.dat" # Use this named file.
XLFSRATCH_12="./data/scratch"    # Relative to current directory.
XLFSRATCH_123="/home/user/data"  # Somewhere besides /tmp.
export XLFRTOPSIS XLFSRATCH_1 XLFSRATCH_12 XLFSRATCH_123

```

Notes:

1. The `XLFSRATCH_number` variable name must be in uppercase, and *number* must not have any leading zeros.
2. `scratch_vars=yes` might be only part of the value for the `XLFRTOPSIS` variable, depending on what other run-time options you have set. See “Setting Run-Time Options” on page 35 for other options that might be part of the `XLFRTOPSIS` value.
3. If the `scratch_vars` run-time option is set to `no` or is undefined or if the applicable `XLFSRATCH_number` variable is not set when the program is run, the program chooses a unique file name for the scratch file and puts it in the directory named by the `TMPDIR` variable or in the `/tmp` directory if the `TMPDIR` variable is not set.

Asynchronous I/O

You may need to use asynchronous I/O for speed and efficiency in scientific programs that perform I/O for large amounts of data. Synchronous I/O blocks the execution of an application until the I/O operation completes. Asynchronous I/O allows an application to continue processing while the I/O operation is performed in the background. You can modify applications to take advantage of the ability to overlap processing and I/O operations. Multiple asynchronous I/O operations can also be performed simultaneously on multiple files that reside on independent devices. For a complete description of the syntax and language elements that you require to use this feature, see the *XL Fortran Language Reference* under the topics:

- **INQUIRE** Statement
- **READ** Statement
- **WAIT** Statement
- **WRITE** Statement

Execution of an Asynchronous Data Transfer Operation

The effect of executing an asynchronous data transfer operation will be as if the following steps were performed in the order specified, with steps (6)-(9) possibly occurring asynchronously:

1. Determine the direction of the data transfer.
2. Identify the unit.
3. Establish the format if one is present.
4. Determine whether an error condition, end-of-file condition, or end-of-record condition has occurred.
5. Cause the variable that you specified in the **IOSTAT=** specifier in the data transfer statement to become defined.
6. Position the file before you transfer data.
7. Transfer data between the file and the entities that you specified by the input/output list (if any).
8. Determine whether an error condition, end-of-file condition, or end-of-record condition has occurred.
9. Position the file after you transfer data.
10. Cause any variables that you specified in the **IOSTAT=** and **SIZE=** specifiers in the **WAIT** statement to become defined.

Usage

You can use Fortran asynchronous **READ** and **WRITE** statements to initiate asynchronous data transfers in Fortran. Execution continues after the asynchronous I/O statement, regardless of whether the actual data transfer has completed.

A program may synchronize itself with a previously initiated asynchronous I/O statement by using a **WAIT** statement. There are two forms of the **WAIT** statement:

1. In a **WAIT** statement without the **DONE=** specifier, the **WAIT** statement halts execution until the corresponding asynchronous I/O statement has completed:

```
integer idvar
integer, dimension(1000):: a
....
READ(unit_number, ID=idvar) a
....
WAIT(ID=idvar)
....
```

2. In a **WAIT** statement with the **DONE=** specifier, the **WAIT** statement returns the completion status of an asynchronous I/O statement:

```

integer idvar
logical done
integer, dimension(1000):: a
....
READ(unit_number, ID=idvar) a
....
WAIT(ID=idvar, DONE=done)
....

```

The variable you specified in the **DONE=** specifier is set to "true" if the corresponding asynchronous I/O statement completes. Otherwise, it is set to "false".

The actual data transfer can take place in the following cases:

- During the asynchronous **READ** or **WRITE** statement
- At any time before the execution of the corresponding **WAIT** statement
- During the corresponding **WAIT** statement

Because of the nature of asynchronous I/O, the actual completion time of the request cannot be predicted.

You specify Fortran asynchronous **READ** and **WRITE** statements by using the **ID=** specifier. The value set for the **ID=** specifier by an asynchronous **READ** or **WRITE** statement must be the same value specified in the **ID=** specifier in the corresponding **WAIT** statement. You must preserve this value until the associated asynchronous I/O statement has completed.

The following program shows a valid asynchronous **WRITE** statement:

```

program sample0
integer, dimension(1000):: a
integer idvar
a = ((i,i=1,1000)/)
WRITE(10, ID=idvar) a
WAIT(ID=idvar)
end

```

The following program is not valid, because XL Fortran destroys the value of the asynchronous I/O identifier before the associated **WAIT** statement:

```

program sample1
integer, dimension(1000):: a
integer idvar
a = ((i,i=1,1000)/)
WRITE(10, ID=idvar) a
idvar = 999 ! Valid id is destroyed.
WAIT(ID=idvar)
end

```

An application that uses asynchronous I/O typically improves performance by overlapping processing with I/O operations. The following is a simple example:

```

program sample2
integer (kind=4), parameter :: isize=1000000, icol=5
integer (kind=4) :: i, j, k
integer (kind=4), dimension(icol) :: handle
integer (kind=4), dimension(isize,icol), static :: a, a1

!
! Opens the file for both synchronous and asynchronous I/O.
!
open(20, form="unformatted", access="direct", &
      status="scratch", recl=isize*4, async="yes")

```

```

!
!   This loop overlaps the initialization of a(:,j) with
!   asynchronous write statements.
!
!   NOTE: The array is written out one column at a time.
!         Since the arrays in Fortran are arranged in column
!         major order, each WRITE statement writes out a
!         contiguous block of the array.
!
!
do 200 j = 1, icol
  a(:,j) = (/ (i*j,i=1, isize) /)
  write(20, id=handle(j), rec=j) a(:,j)
200 end do

!
!   Wait for all writes to complete before reading.
!
do 300 j = 1, icol
  wait(id=handle(j))
300 end do

!
!   Reads in the first record.
!
read(20, id=handle(1), rec=1) a1(:,1)

do 400 j = 2, icol
  k = j - 1
!
!   Waits for a previously initiated read to complete.
!
  wait(id=handle(k))
!
!   Initiates the next read immediately.
!
  read(20, id=handle(j), rec=j) a1(:,j)
!
!   While the next read is going on, we do some processing here.
!
  do 350 i = 1, isize
    if (a(i,k) .ne. a1(i,k)) then
      print *, "(i,k) &
        & expected ", a(i,k), " got ", a1(i,k)
    end if
350   end do
400 end do

!
!   Finish the last record.
!
wait(id=handle(icol))

do 450 i = 1, isize
  if (a(i,icol) .ne. a1(i,icol)) then
    print *, "(i,icol) &
      & expected ", a(i,icol), " got ", a1(i,icol)
  end if
450 end do

close(20)
end

```

Performance

To maximize the benefits of asynchronous I/O, you should only use it for large contiguous data items.

It is possible to perform asynchronous I/O on a large number of small items, but the overall performance will suffer. This is because extra processing overhead is required to maintain each item for asynchronous I/O. Performing asynchronous I/O on a larger number of small items is strongly discouraged. The following are two examples:

1. WRITE(unit_number, ID=idvar) a1(1:100000000:2)
2. WRITE(unit_number, ID=idvar) (a2(i,j),j=1,100000000)

Performing asynchronous I/O on unformatted sequential files is less efficient. This is because each record might have a different length, and these lengths are stored with the records themselves. You should use unformatted direct access or unformatted stream access, if possible, to maximize the benefits of asynchronous I/O.

Compiler-Generated Temporary I/O Items

There are situations when the compiler must generate a temporary variable to hold the result of an I/O item expression. In such cases, synchronous I/O is performed on the temporary variable, regardless of the mode of transfer that you specified in the I/O statement. The following are examples of such cases:

1. For **READ**, when an array with vector subscripts appears as an input item:

- a. integer a(5), b(3)

 b = (/1,3,5/)

 read(99, id=i) a(b)
- b. real a(10)

 read(99, id=i) a(/1,3,5/)

2. For **WRITE**, when an output item is an expression that is a constant or a constant of certain derived types:

- a. write(99, id=i) 1000
- b. integer a

 parameter(a=1000)

 write(99, id=i) a

```

c.      type mytype
         integer a
         integer b
         end type mytype

         write(99,id=i) mytype(4,5)

```

3. For **WRITE**, when an output item is a temporary variable:

```

a.      write(99,id=i) 99+100
b.      write(99,id=i) a+b
c.      external ff
         real(8) ff

```

```

         write(99,id=i) ff()

```

4. For **WRITE**, when an output item is an expression that is an array constructor:

```

write(99,id=i) (/1,2,3,4,5/)

```

5. For **WRITE**, when an output item is an expression that is a scalarized array:

```

integer a(5),b(5)
write(99,id=i) a+b

```

Error Handling

For an asynchronous data transfer, errors or end-of-file conditions might occur either during execution of the data transfer statement or during subsequent data transfer. If these conditions do not result in the termination of the program, you can detect these conditions via **ERR=**, **END=** and **IOSTAT=** specifiers in the data transfer or in the matching **WAIT** statement.

Execution of the program terminates if an error condition occurs during execution or during subsequent data transfer of an input/output statement that contains neither an **IOSTAT=** nor an **ERR=** specifier. In the case of a recoverable error, if the **IOSTAT=** and **ERR=** specifiers are not present, the program terminates if you set the **err_recovery** run-time option to **no**. If you set the **err_recovery** run-time option to **yes**, recovery action occurs, and the program continues.

If an asynchronous data transfer statement causes either of the following events, a matching **WAIT** statement cannot run, because the **ID=** value is not defined:

- A branch to the label that you specified by **ERR=** or **END=**
- The **IOSTAT=** specifier to be set to a non-zero value

XL Fortran Thread-Safe I/O Library

The XL Fortran thread-safe I/O library **libxlf90_r.so** provides support for parallel execution of Fortran I/O statements. For Fortran programs that contain I/O statements in a parallelized loop or that create multiple threads and execute I/O statements from different threads at the same time, you must use this library. In other words, to perform Fortran I/O in parallel, you must link applications with this library to get expected results.

Synchronization of I/O Operations

During parallel execution, multiple threads might perform I/O operations on the same file at the same time. If they are not synchronized, the results of these I/O operations could be shuffled or merged or both, and the application might produce incorrect results or even terminate. The XL Fortran thread-safe I/O library synchronizes I/O operations for parallel applications. It performs the synchronization within the I/O library, and it is transparent to application programs. The purpose of the synchronization is to ensure the integrity and correctness of each individual I/O operation. However, the thread-safe I/O library

does not have control over the order in which threads execute I/O statements. Therefore, the order of records read in or written out is not predictable under parallel I/O operations. Refer to “Parallel I/O Issues” for details.

External Files: For external files, the synchronization is performed on a per-unit basis. The XL Fortran thread-safe I/O library ensures that only one thread can access a particular logical unit to prevent several threads from interfering with each other. When a thread is performing an I/O operation on a unit, other threads attempting to perform I/O operations on the same unit must wait until the first thread finishes its operation. Therefore, the execution of I/O statements by multiple threads on the same unit is serialized. However, the thread-safe I/O library does not prevent threads from operating on different logical units in parallel. In other words, parallel access to different logical units is not necessarily serialized.

Functionality of I/O under Synchronization: The XL Fortran thread-safe I/O library sets its internal locks to synchronize access to logical units. This should not have any functional impact on the I/O operations performed by a Fortran program. Also, it will not impose any additional restrictions to the operability of Fortran I/O statements except for the use of I/O statements in a signal handler that is invoked asynchronously. Refer to “Use of I/O Statements in Signal Handlers” on page 299 for details.

The Fortran standard prohibits a function reference from appearing in an expression anywhere in an I/O statement if such a reference causes another I/O statement to run. This restriction still applies with the XL Fortran thread-safe I/O library.

Parallel I/O Issues

The order in which parallel threads perform I/O operations is not predictable. The XL Fortran thread-safe I/O library does not have control over the ordering. It will allow whichever thread that executes an I/O statement on a particular logical unit and obtains the lock on it first to proceed with the operation. Therefore, only use parallel I/O in cases where at least one of the following is true:

- Each thread performs I/O on a predetermined record in direct-access files.
- Each thread performs I/O on a different part of a stream-access file. Different I/O statements cannot use the same, or overlapping, areas of a file.
- The result of an application does not depend on the order in which records are written out or read in.
- Each thread performs I/O on a different file.

In these cases, results of the I/O operations are independent of the order in which threads execute. However, you might not get the performance improvements that you expect, since the I/O library serializes parallel access to the same logical unit from multiple threads. Examples of these cases are as follows:

- Each thread performs I/O on a pre-determined record in a direct-access file:

```
do i = 1, 10
  write(4, '(i4)', rec = i) a(i)
enddo
```

- Each thread performs I/O on a different part of a stream-access file. Different I/O statements cannot use the same, or overlapping, areas of a file.

```
do i = 1, 9
  write(4, '(i4)', pos = 1 + 5 * (i - 1)) a(i)
  ! We use 5 above because i4 takes 4 file storage
  ! units + 1 file storage unit for the record marker.
enddo
```

- In the case that each thread operates on a different file, since threads share the status of the logical units connected to the files, the thread still needs to obtain the lock on the logical unit for either retrieving or updating the status of the logical unit. However, the thread-safe I/O library allows threads to perform the data transfer between the logical unit and the I/O list item in parallel. If an application contains a large number of small I/O requests in a parallel region, you might not get the expected performance because of the lock contention. Consider the following example:

```

program example

  use omp_lib

  integer, parameter :: num_of_threads = 4, max = 5000000
  character*10 file_name
  integer i, file_unit, thread_id
  integer, dimension(max, 2 * num_of_threads) :: aa

  call omp_set_num_threads(num_of_threads)

!$omp parallel private(file_name, thread_id, file_unit, i) shared(aa)

  thread_id = omp_get_thread_num()
  file_name = 'file_'
  file_name(6:6) = char(ichar('0') + thread_id)
  file_unit = 10 + thread_id

  open(file_unit, file = file_name, status = 'old', action = 'read')

  do i = 1, max
    read(file_unit, *) aa(i, thread_id * 2 + 1), aa(i, thread_id * 2 + 2)
  end do

  close(file_unit)

!$omp end parallel
end

```

The I/O library synchronizes retrieving and updating the status of the logical units while performing data transfer in parallel. In order to maximize the parallelism the I/O library provides, it is recommended to increase the size of data transfer per I/O request. The do loop, therefore, should be rewritten as follows:

- ```

 read(file_unit, *) a(:, thread_id * 2 + 1 : thread_id * 2 + 2)

 do i = 1, max
 ! Do something for each element of array 'aa'.
 end do

```
- The result does not depend on the order in which records are written out or read in:

```

real a(100)
do i = 1, 10
 read(4) a(i)
enddo
call qsort_(a)

```
  - Each thread performs I/O on a different logical unit of direct access, sequential access, or stream access:

```

do i = 11, 20
 write(i, '(i4)') a(i - 10)
enddo

```

For multiple threads to write to or read from the same sequential-access file, or to write to or read from the same stream-access file without using the **POS=** specifier,

the order of records written out or read in depends on the order in which the threads execute the I/O statement on them. This order, as stated previously, is not predictable. Therefore, the result of an application could be incorrect if it assumes records are sequentially related and cannot be arbitrarily written out or read in. For example, if the following loop is parallelized, the numbers printed out will no longer be in the sequential order from 1 to 500 as the result of a serial execution:

```
do i = 1, 500
 print *, i
enddo
```

Applications that depend on numbers being strictly in the specified order will not work correctly.

The XL Fortran run-time option **multconn=yes** allows connection of the same file to more than one logical unit simultaneously. Since such connections can only be made for reading (**ACCESS='READ'**), access from multiple threads to logical units that are connected to the same file will produce predictable results.

## Use of I/O Statements in Signal Handlers

There are basically two kinds of signals in the POSIX signal model: *synchronously* and *asynchronously* generated signals. Signals caused by the execution of some code of a thread, such as a reference to an unmapped, protected, or bad memory (**SIGSEGV** or **SIGBUS**), floating-point exception (**SIGFPE**), execution of a trap instruction (**SIGTRAP**), or execution of illegal instructions (**SIGILL**) are said to be synchronously generated. Signals may also be generated by events outside the process: for example, **SIGINT**, **SIGHUP**, **SIGQUIT**, **SIGIO**, and so on. Such events are referred to as interrupts. Signals that are generated by interrupts are said to be asynchronously generated.

The XL Fortran thread-safe I/O library is asynchronous signal unsafe. This means that the XL Fortran I/O statements cannot be used in a signal handler that is entered because of an asynchronously generated signal. The behavior of the system is undefined when an XL Fortran I/O statement is called from a signal handler that interrupts an I/O statement. However, it is safe to use I/O statements in signal handlers for synchronous signals.

Sometimes an application can guarantee that a signal handler is not entered asynchronously. For example, an application might mask signals except when it runs certain known sections of code. In such situations, the signal will not interrupt any I/O statements and other asynchronous signal unsafe functions. Therefore, you can still use Fortran I/O statements in an asynchronous signal handler.

A much easier and safer way to handle asynchronous signals is to block signals in all threads and to explicitly wait (using **sigwait()**) for them in one or more separate threads. The advantage of this approach is that the **handler** thread can use Fortran I/O statements as well as other asynchronous signal unsafe routines.

## Asynchronous Thread Cancellation

When a thread enables asynchronous thread cancellability, any cancellation request is acted upon immediately. The XL Fortran thread-safe I/O library is not asynchronous thread cancellation safe. The behavior of the system is undefined if a thread is cancelled asynchronously while it is in the XL Fortran thread-safe I/O library.



---

## Interlanguage Calls

This section gives details about performing interlanguage calls from your Fortran program: that is, calling routines that were written in a language other than Fortran. It assumes that you are familiar with the syntax of all applicable languages.

---

### Conventions for XL Fortran External Names

To assist you in writing mixed-language programs, XL Fortran follows a consistent set of rules when translating the name of a global entity into an external name that the linker can resolve:

- Both the underscore (`_`) and the dollar sign (`$`) are valid characters anywhere in names.

Because names that begin with an underscore are reserved for the names of library routines, do not use an underscore as the first character of a Fortran external name.

To avoid conflicts between Fortran and non-Fortran function names, you can compile the Fortran program with the `-qextname` option. This option adds an underscore to the end of the Fortran names. Then use an underscore as the last character of any non-Fortran procedures that you want to call from Fortran.

- Names can be up to 250 characters long.
- Program and symbolic names are interpreted as all lowercase by default. If you are writing new non-Fortran code, use all-lowercase procedure names to simplify calling the procedures from Fortran.

You can use the `-U` option or the `@PROCESS MIXED` directive if you want the names to use both uppercase and lowercase:

```
@process mixed
 external C_Func ! With MIXED, we can call C_Func, not just c_func.
 integer aBC, ABC ! With MIXED, these are different variables.
 common /xYz/ aBc ! The same applies to the common block names.
 common /XYZ/ ABC ! xYz and XYZ are external names that are
 ! visible during linking.
end
```

- Names for module procedures are formed by concatenating `__` (two underscores), the module name, `_IMOD_` (for intrinsic modules) or `_NMOD_` (for non-intrinsic modules), and the name of the module procedure. For example, module procedure `MYPROC` in module `MYMOD` has the external name `__mymod_NMOD_myproc`.

**Note:** Symbolic debuggers and other tools should account for this naming scheme when debugging XL Fortran programs that contain module procedures. For example, some debuggers default to lowercase for program and symbolic names. This behavior should be changed to use mixed case when debugging XL Fortran programs with module procedures.

- The XL compilers generate code that uses `main` as an external entry point name. You can only use `main` as an external name in these contexts:
  - A Fortran program or local-variable name. (This restriction means that you cannot use `main` for the name of an external function, external subroutine, block data program unit, or common block. References to such an object use the compiler-generated `main` instead of your own.)
  - The name of the top-level main function in a C program.
- Some other potential naming conflicts may occur when linking a program. For instructions on avoiding them, see “Avoiding Naming Conflicts during Linking” on page 33.

If you are porting your application from another system and your application does encounter naming conflicts like these, you may need to use the “-qextname Option” on page 129.

---

## Mixed-Language Input and Output

To improve performance, the XL Fortran run-time library has its own buffers and its own handling of these buffers. This means that mixed-language programs cannot freely mix I/O operations on the same file from the different languages. To maintain data integrity in such cases:

- If the file position is not important, open and explicitly close the file within the Fortran part of the program before performing any I/O operations on that file from subprograms written in another language.
- To open a file in Fortran and manipulate the open file from another language, call the `flush_` procedure to save any buffer for that file, and then use the `getfd` procedure to find the corresponding file descriptor and pass it to the non-Fortran subprogram. As an alternative to calling the `flush_` procedure, you can use the `buffering` run-time option to disable the buffering for I/O operations. When you specify `buffering=disable_preconn`, XL Fortran disables the buffering for preconnected units. When you specify `buffering=disable_all`, XL Fortran disables the buffering for all logical units.

**Note:** After you call `flush_` to flush the buffer for a file, do not do anything to the file from the Fortran part of the program except to close it when the non-Fortran processing is finished.

- If any XL Fortran subprograms containing `WRITE` statements are called from a non-Fortran main program, explicitly `CLOSE` the data file, or use the `flush_` subroutine in the XL Fortran subprograms to ensure that the buffers are flushed. Alternatively, you can use the `buffering` run-time option to disable buffering for I/O operations.

**Related Information:** For more information on the `flush_` and `getfd` procedures, see the *Service and Utility Procedures* section in the *XL Fortran Language Reference*. For more information on the `buffering` run-time option, see “Setting Run-Time Options” on page 35.

---

## Mixing Fortran and C++

Most of the information in this section applies to Fortran and C — languages with similar data types and naming schemes. However, to mix Fortran and C++ in the same program, you must add an extra level of indirection and pass the interlanguage calls through C++ wrapper functions.

Because the C++ compiler mangles the names of some C++ objects, you must use a C++ compiler to link the final program and include `-L` and `-I` options for the XL Fortran library directories and libraries.

```

program main

integer idim,idim1

idim = 35
idim1= 45

write(6,*) 'Inside Fortran calling first C function'
call cfun(idim)
write(6,*) 'Inside Fortran calling second C function'
call cfun1(idim1)
write(6,*) 'Exiting the Fortran program'
end

```

*Figure 1. Main Fortran Program That Calls C++ (main1.f)*

```

#include <stdio.h>
#include "cplus.h"

extern "C" void cfun(int *idim);
extern "C" void cfun1(int *idim1);

void cfun(int *idim){
 printf("%dInside C function before creating C++ Object\n");
 int i = *idim;
 junk<int>* jj= new junk<int>(10,30);
 jj->store(idim);
 jj->print();
 printf("%dInside C function after creating C++ Object\n");
 delete jj;
 return;
}

void cfun1(int *idim1) {
 printf("%dInside C function cfun1 before creating C++ Object\n");
 int i = *idim1;
 temp<double> *tmp = new temp<double>(40, 50.54);
 tmp->print();
 printf("%dInside C function after creating C++ temp object\n");
 delete tmp;
 return;
}

```

*Figure 2. C++ Wrapper Functions for Calling C++ (cfun.C)*

```

#include <iostream.h>

template<class T> class junk {

private:
 int inter;
 T templ_mem;
 T stor_val;

public:
 junk(int i,T j): inter(i),templ_mem(j)
 {cout <<"***Inside C++ constructor" << endl;}

 ~junk() {cout <<"***Inside C++ Destructor" << endl;}

 void store(T *val){ stor_val = *val;}

 void print(void) {cout << inter << "\t" << templ_mem ;
 cout <<"\t" << stor_val << endl; }};

template<class T> class temp {

private:
 int internal;
 T temp_var;

public:
 temp(int i, T j): internal(i),temp_var(j)
 {cout <<"***Inside C++ temp Constructor" <<endl;}

 ~temp() {cout <<"***Inside C++ temp destructor" <<endl;}

 void print(void) {cout << internal << "\t" << temp_var << endl;}};

```

Figure 3. C++ Code Called from Fortran (cplus.h)

Compiling this program, linking it with the `x1C` or `g++` command, and running it produces this output:

```

Inside Fortran calling first C function
%Inside C function before creating C++ Object
***Inside C++ constructor
10 30 35
%Inside C function after creating C++ Object
***Inside C++ Destructor
Inside Fortran calling second C function
%Inside C function cfun1 before creating C++ Object
***Inside C++ temp Constructor
40 50.54
%Inside C function after creating C++ temp object
***Inside C++ temp destructor
Exiting the Fortran program

```

---

## Making Calls to C Functions Work

When you pass an argument to a subprogram call, the usual Fortran convention is to pass the address of the argument. Many C functions expect arguments to be passed as values, however, not as addresses. For these arguments, specify them as `%VAL(argument)` in the call to C, as follows:

```

MEMBLK = MALLOC(1024) ! Wrong, passes the address of the constant
MEMBLK = MALLOC(N) ! Wrong, passes the address of the variable

MEMBLK = MALLOC(%VAL(1024)) ! Right, passes the value 1024
MEMBLK = MALLOC(%VAL(N)) ! Right, passes the value of the variable

```

See “Passing Arguments By Reference or By Value” on page 309 and %VAL and %REF in the *XL Fortran Language Reference* for more details.

## Passing Data From One Language to Another

The following table shows the data types available in the XL Fortran and C languages. This section shows how Fortran arguments can be passed by reference to C programs. To use the Fortran 2003 Draft Standard interoperability features, such as the **BIND(C)** attribute and **ISO\_C\_BINDING** module support, see the *XL Fortran Language Reference*.

### Passing Arguments between Languages

Table 19. Corresponding Data Types in Fortran and C. When calling Fortran, the C routines must pass arguments as pointers to the types listed in this table.

| XL Fortran Data Types                                                                                                                                                                                   | C Data Types                             |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------|
| INTEGER(1), BYTE                                                                                                                                                                                        | signed char                              |
| INTEGER(2)                                                                                                                                                                                              | signed short                             |
| INTEGER(4)                                                                                                                                                                                              | signed int                               |
| INTEGER(8)                                                                                                                                                                                              | signed long long                         |
| REAL, REAL(4)                                                                                                                                                                                           | float                                    |
| REAL(8), DOUBLE PRECISION                                                                                                                                                                               | double                                   |
| REAL(16)                                                                                                                                                                                                | long double (see note 1)                 |
| COMPLEX, COMPLEX(8)                                                                                                                                                                                     | _Complex float (see note 2)              |
| COMPLEX(16), DOUBLE COMPLEX                                                                                                                                                                             | _Complex double (see note 2)             |
| COMPLEX(32)                                                                                                                                                                                             | _Complex long double (see notes 1 and 2) |
| LOGICAL(1)                                                                                                                                                                                              | unsigned char                            |
| LOGICAL(2)                                                                                                                                                                                              | unsigned short                           |
| LOGICAL(4)                                                                                                                                                                                              | unsigned int                             |
| LOGICAL(8)                                                                                                                                                                                              | unsigned long long                       |
| CHARACTER                                                                                                                                                                                               | char                                     |
| CHARACTER(n)                                                                                                                                                                                            | char[n]                                  |
| Integer POINTER                                                                                                                                                                                         | void *                                   |
| Array                                                                                                                                                                                                   | array                                    |
| Sequence-derived type                                                                                                                                                                                   | structure (with a C packed structure)    |
| <b>Notes:</b><br>1. Requires a C compiler that supports 128-bit long double. XL C does not officially support <b>-qldb128</b> on Linux. Use with discretion.<br>2. Requires a C99-compliant C compiler. |                                          |

#### Notes:

1. In interlanguage communication, it is often necessary to use the %VAL and %REF built-in functions that are defined in “Passing Arguments By Reference or By Value” on page 309.
2. C programs automatically convert float values to double and short integer values to integer when calling an unprototyped C function. Because XL Fortran does not perform a conversion on **REAL(4)** quantities passed by value, you

should not pass **REAL(4)** and **INTEGER(2)** values as arguments to C functions that you have not declared with function prototypes.

3. The Fortran-derived type and the C structure must match in the number, data type, and length of subobjects to be compatible data types.

**Related Information:** One or more sample programs under the directory **/opt/ibmcmp/xlf/9.1/samples** illustrate how to call from Fortran to C.

To use the Fortran 2003 Draft Standard interoperability features provided by XL Fortran, see the *Language Interoperability Features* section in the *XL Fortran Language Reference*.

## Passing Global Variables between Languages

To access a C data structure from within a Fortran program or to access a common block from within a C program, follow these steps:

1. Create a named common block that provides a one-to-one mapping of the C structure members. If you have an unnamed common block, change it to a named one. Name the common block with the name of the C structure.
2. Declare the C structure as a global variable by putting its declaration outside any function or inside a function with the **extern** qualifier.
3. Compile the C source file to get packed structures.

```
program cstruct struct mystuff {
real(8) a,d double a;
integer b,c int b,c;
. double d;
. };
common /mystuff/ a,b,c,d
.
.
end main() {
 }
```

If you do not have a specific need for a named common block, you can create a sequence-derived type with the same one-to-one mapping as a C structure and pass it as an argument to a C function. You must compile the C source file to get packed structures.

Common blocks that are declared **THREADLOCAL** are thread-specific data areas that are dynamically allocated by compiler-generated code. A static block is still reserved for a **THREADLOCAL** common block, but the compiler and the compiler's run-time environment use it for control information. If you need to share **THREADLOCAL** common blocks between Fortran and C procedures, your C source must be aware of the implementation of the **THREADLOCAL** common block. For more information, see **THREADLOCAL** common blocks, the *Directives* section in the *XL Fortran Language Reference*, and Appendix A, "Sample Fortran Programs," on page 345.

Common blocks that are declared **THREADPRIVATE** can be accessed using a C global variable that is declared as **THREADPRIVATE**.

## Passing Character Types between Languages

One difficult aspect of interlanguage calls is passing character strings between languages. The difficulty is due to the following underlying differences in the way that different languages represent such entities:

- The only character type in Fortran is **CHARACTER**, which is stored as a set of contiguous bytes, one character per byte. The length is not stored as part of the entity. Instead, it is passed by value as an extra argument at the end of the declared argument list when the entity is passed as an argument.
- Character strings in C are stored as arrays of the type **char**. A null character indicates the end of the string.

**Note:** To have the compiler automatically add the null character to certain character arguments, you can use the “-qnullterm Option” on page 171.

If you are writing both parts of the mixed-language program, you can make the C routines deal with the extra Fortran length argument, or you can suppress this extra argument by passing the string using the **%REF** function. If you use **%REF**, which you typically would for pre-existing C routines, you need to indicate where the string ends by concatenating a null character to the end of each character string that is passed to a C routine:

```
! Initialize a character string to pass to C.
 character*6 message1 /'Hello\0'/
! Initialize a character string as usual, and append the null later.
 character*5 message2 /'world'/

! Pass both strings to a C function that takes 2 (char *) arguments.
 call cfunc(%ref(message1), %ref(message2 // '\0'))
 end
```

For compatibility with C language usage, you can encode the following escape sequences in XL Fortran character strings:

*Table 20. Escape Sequences for Character Strings*

| Escape | Meaning                                                                     |
|--------|-----------------------------------------------------------------------------|
| \b     | Backspace                                                                   |
| \f     | Form feed                                                                   |
| \n     | New-line                                                                    |
| \r     | Carriage return                                                             |
| \t     | Tab                                                                         |
| \0     | Null                                                                        |
| \'     | Apostrophe (does not terminate a string)                                    |
| \"     | Double quotation mark (does not terminate a string)                         |
| \\     | Backslash                                                                   |
| \x     | <i>x</i> , where <i>x</i> is any other character (the backslash is ignored) |

If you do not want the backslash interpreted as an escape character within strings, you can compile with the **-qnoescape** option.

## Passing Arrays between Languages

Fortran stores array elements in ascending storage units in column-major order. C stores array elements in row-major order. Fortran array indexes start at 1, while C array indexes start at 0.

The following example shows how a two-dimensional array that is declared by A(3,2) is stored in Fortran and C:

Table 21. Corresponding Array Layouts for Fortran and C. The Fortran array reference A(X,Y,Z) can be expressed in C as a[Z-1][Y-1][X-1]. Keep in mind that although C passes individual scalar array elements by value, it passes arrays by reference.

|                      | Fortran Element Name | C Element Name |
|----------------------|----------------------|----------------|
| Lowest storage unit  | A(1,1)               | A[0][0]        |
|                      | A(2,1)               | A[0][1]        |
|                      | A(3,1)               | A[1][0]        |
|                      | A(1,2)               | A[1][1]        |
|                      | A(2,2)               | A[2][0]        |
| Highest storage unit | A(3,2)               | A[2][1]        |

To pass all or part of a Fortran array to another language, you can use Fortran 90/Fortran 95 array notation:

```
REAL, DIMENSION(4,8) :: A, B(10)

! Pass an entire 4 x 8 array.
CALL CFUNC(A)
! Pass only the upper-left quadrant of the array.
CALL CFUNC(A(1:2,1:4))
! Pass an array consisting of every third element of A.
CALL CFUNC(A(1:4:3,1:8))
! Pass a 1-dimensional array consisting of elements 1, 2, and 4 of B.
CALL CFUNC(B((/1,2,4/)))
```

Where necessary, the Fortran program constructs a temporary array and copies all the elements into contiguous storage. In all cases, the C routine needs to account for the column-major layout of the array.

Any array section or noncontiguous array is passed as the address of a contiguous temporary unless an explicit interface exists where the corresponding dummy argument is declared as an assumed-shape array or a pointer. To avoid the creation of array descriptors (which are not supported for interlanguage calls) when calling non-Fortran procedures with array arguments, either do not give the non-Fortran procedures any explicit interface, or do not declare the corresponding dummy arguments as assumed-shape or pointers in the interface:

```
! This explicit interface must be changed before the C function
! can be called.
INTERFACE
 FUNCTION CFUNC (ARRAY, PTR1, PTR2)
 INTEGER, DIMENSION (:) :: ARRAY ! Change this : to *.
 INTEGER, POINTER, DIMENSION (:) :: PTR1 ! Change this : to *
 ! and remove the POINTER
 ! attribute.
 REAL, POINTER :: PTR2 ! Remove this POINTER
 ! attribute or change to TARGET.
 END FUNCTION
END INTERFACE
```

## Passing Pointers between Languages

Integer **POINTERS** always represent the address of the pointee object and must be passed by value:

```
CALL CFUNC(%VAL(INTPTR))
```

Fortran 90 **POINTERS** can also be passed back and forth between languages but only if there is no explicit interface for the called procedure or if the argument in

the explicit interface does not have a **POINTER** attribute or assumed-shape declarator. You can remove any **POINTER** attribute or change it to **TARGET** and can change any deferred-shape array declarator to be explicit-shape or assumed-size.

Because of XL Fortran's call-by-reference conventions, you must pass even scalar values from another language as the address of the value, rather than the value itself. For example, a C function passing an integer value *x* to Fortran must pass `&x`. Also, a C function passing a pointer value *p* to Fortran so that Fortran can use it as an integer **POINTER** must declare it as `void **p`. A C array is an exception: you can pass it to Fortran without the `&` operator.

## Passing Arguments By Reference or By Value

To call subprograms written in languages other than Fortran (for example, user-written C programs, or operating system routines), the actual arguments may need to be passed by a method different from the default method used by Fortran. C routines, including those in system libraries such as **libc.so**, require you to pass arguments by value instead of by reference. (Although C passes individual scalar array elements by value, it passes arrays by reference.)

You can change the default passing method by using the **%VAL** and **%REF** built-in functions in the argument list of a **CALL** statement or function reference. You cannot use them in the argument lists of Fortran procedure references or with alternate return specifiers.

**%REF** Passes an argument by reference (that is, the called subprogram receives the address of the argument). It is the same as the default calling method for Fortran except that it also suppresses the extra length argument for character strings.

**%VAL** Passes an argument by value (that is, the called subprogram receives an argument that has the same value as the actual argument, but any change to this argument does not affect the actual argument).

You can use this built-in function with actual arguments that are **CHARACTER(1)**, **BYTE**, logical, integer, real, or complex expressions or that are sequence-derived type. Objects of derived type cannot contain pointers, arrays, or character structure components whose lengths are greater than one byte.

You cannot use **%VAL** with actual arguments that are array entities, procedure names, or character expressions of length greater than one byte.

**%VAL** causes XL Fortran to pass the actual argument as 32-bit or 64-bit intermediate values.

### In 32-bit Mode

If the actual argument is one of the following:

- An integer or a logical that is shorter than 32 bits, it is sign-extended to a 32-bit value.
- An integer or a logical that is longer than 32 bits, it is passed as two 32-bit intermediate values.
- Of type real or complex, it is passed as multiple 64-bit intermediate values.
- Of sequence-derived type, it is passed as multiple 32-bit intermediate values.

Byte-named constants and variables are passed as if they were **INTEGER(1)**. If the actual argument is a **CHARACTER(1)**, the compiler pads it on the left with zeros to a 32-bit value, regardless of whether you specified the **-qctyplss** compiler option.

### In 64-bit Mode

If the actual argument is one of the following:

- An integer or a logical that is shorter than 64 bits, it is sign-extended to a 64-bit value.
- Of type real or complex, it is passed as multiple 64-bit intermediate values.
- Of sequence-derived type, it is passed as multiple 64-bit intermediate values.

Byte-named constants and variables are passed as if they were **INTEGER(1)**. If the actual argument is a **CHARACTER(1)**, the compiler pads it on the left with zeros to a 64-bit value, regardless of whether you specified the **-qctyplss** compiler option.

If you specified the **-qautodbl** compiler option, any padded storage space is not passed except for objects of derived type.

```
EXTERNAL FUNC
COMPLEX XVAR
IVARB=6

CALL RIGHT2(%REF(FUNC)) ! procedure name passed by reference
CALL RIGHT3(%VAL(XVAR)) ! complex argument passed by value
CALL TPROG(%VAL(IVARB)) ! integer argument passed by value
END
```

### Explicit Interface for %VAL and %REF

You can specify an explicit interface for non-Fortran procedures to avoid coding calls to **%VAL** and **%REF** in each argument list, as follows:

```
INTERFACE
 FUNCTION C_FUNC(%VAL(A),%VAL(B)) ! Now you can code "c_func(a,b)"
 INTEGER A,B ! instead of
 END FUNCTION C_FUNC ! "c_func(%val(a),%val(b))".
END INTERFACE
```

## Passing Complex Values to/from gcc

Passing complex values between Fortran and Gnu C++ depends on what is specified for the `-qfloat=[no]complexgcc` suboption. If `-qfloat=complexgcc` is specified, the compiler uses Linux conventions when passing or returning complex numbers. `-qfloat=nocomplexgcc` is the default.

For `-qfloat=complexgcc` in 32-bit mode, the compiler passes `COMPLEX *8` values in 2 general-purpose registers (GPRs) and `COMPLEX *16` values in 4 GPRs. In 64-bit mode, `COMPLEX *8` values are passed in 1 GPR, and `COMPLEX *16` in 2 GPRs. For `-qfloat=nocomplexgcc`, `COMPLEX *8` and `COMPLEX *16` values are passed in 2 floating-point registers (FPRs). `COMPLEX *32` values are always passed in 4 FPRs for both `-qfloat=complexgcc` and `-qfloat=nocomplexgcc` (since gcc does not support `COMPLEX*32`).

For `-qfloat=complexgcc` in 32-bit mode, `COMPLEX *8` values are returned in GPR3-GPR4, and `COMPLEX *16` in GPR3-GPR6. In 64-bit mode, `COMPLEX *8` values are returned in GPR3, and `COMPLEX *16` in GPR 3-GPR4. For `-qfloat=nocomplexgcc`, `COMPLEX *8` and `COMPLEX *16` values are returned in FPR1-FPR2. For both `-qfloat=complexgcc` and `-qfloat=nocomplexgcc`, `COMPLEX *32` is always returned in FPR1-FPR4.

## Returning Values from Fortran Functions

XL Fortran does not support calling certain types of Fortran functions from non-Fortran procedures. If a Fortran function returns a pointer, array, or character of nonconstant length, do not call it from outside Fortran.

You can call such a function indirectly:

```
SUBROUTINE MAT2(A,B,C) ! You can call this subroutine from C, and the
 ! result is stored in C.
INTEGER, DIMENSION(10,10) :: A,B,C
C = ARRAY_FUNC(A,B) ! But you could not call ARRAY_FUNC directly.
END
```

## Arguments with the OPTIONAL Attribute

When you pass an optional argument by reference, the address in the argument list is zero if the argument is not present.

When you pass an optional argument by value, the value is zero if the argument is not present. The compiler uses an extra register argument to differentiate that value from a regular zero value. If the register has the value 1, the optional argument is present; if it has the value 0, the optional argument is not present.

**Related Information:** See “Order of Arguments in Argument List” on page 320.

## Arguments with the INTENT Attribute

Currently, declaring arguments with the `INTENT` attribute does not change the linkage convention for a procedure. However, because this part of the convention is subject to change in the future, we recommend not calling from non-Fortran procedures into Fortran procedures that have `INTENT(IN)` arguments.

## Type Encoding and Checking

Run-time errors are hard to find, and many of them are caused by mismatched procedure interfaces or conflicting data definitions. Therefore, it is a good idea to find as many of these problems as possible at compile or link time.

## Assembler-Level Subroutine Linkage Conventions

The subroutine linkage convention specifies the machine state at subroutine entry and exit, allowing routines that are compiled separately in the same or different languages to be linked. The information on subroutine linkage and system calls in the *System V Application Binary Interface: PowerPC Processor Supplement* and *64-bit PowerPC ELF Application Binary Interface Supplement* are the base references on this topic. You should consult these for full details. This section summarizes the information needed to write mixed-language Fortran and assembler programs or to debug at the assembler level, where you need to be concerned with these kinds of low-level details.

The system linkage convention passes arguments in registers, taking full advantage of the large number of floating-point registers (FPRs) and general-purpose registers (GPRs) and minimizing the saving and restoring of registers on subroutine entry and exit. The linkage convention allows for argument passing and return values to be in FPRs, GPRs, or both.

The following table lists floating-point registers and their functions. The floating-point registers are double precision (64 bits).

*Table 22. Floating-Point Register Usage across Calls*

| Register | Preserved Across Calls | Use                                          |
|----------|------------------------|----------------------------------------------|
| 0        | no                     |                                              |
| 1        | no                     | FP parameter 1, function return 1.           |
| 2        | no                     | FP parameter 2, function return 2.           |
| 3        | no                     | FP parameter 3, function return complex *32. |
| 4        | no                     | FP parameter 4, function return complex *32. |
| ⋮        | ⋮                      | ⋮                                            |
| 8        | no                     | FP parameter 8                               |
| 9-13     | no                     |                                              |
| 14-31    | yes                    | local variables                              |

The following table lists general-purpose registers and their functions.

*Table 23. General-Purpose Register Usage across Calls*

| Register | Preserved Across Calls | Use                                   |
|----------|------------------------|---------------------------------------|
| 0        | no                     |                                       |
| 1        | yes                    | Stack pointer.                        |
| 2        | yes                    | System-reserved.                      |
| 3        | no                     | 1st word of arg list; return value 1. |
| 4        | no                     | 2nd word of arg list; return value 2. |
| 5        | no                     | 3rd word of arg list                  |
| ⋮        | ⋮                      | ⋮                                     |
| 10       | no                     | 8th word of arg list.                 |

Table 23. General-Purpose Register Usage across Calls (continued)

| Register                                                                                                                                                                                                                                                                                                                                                        | Preserved Across Calls | Use                                        |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------|--------------------------------------------|
| 11-12                                                                                                                                                                                                                                                                                                                                                           | no                     |                                            |
| 13                                                                                                                                                                                                                                                                                                                                                              | yes                    | SDA pointer.                               |
| 14-30                                                                                                                                                                                                                                                                                                                                                           | no                     | Local variables.                           |
| 31                                                                                                                                                                                                                                                                                                                                                              | yes                    | Local variables or “environment pointers”. |
| If a register is not designated as preserved, its contents may be changed during the call, and the caller is responsible for saving any registers whose values are needed later. Conversely, if a register is supposed to be preserved, the callee is responsible for preserving its contents across the call, and the caller does not need any special action. |                        |                                            |

The following table lists special-purpose register conventions.

Table 24. Special-Purpose Register Usage across Calls

| Register                 | Preserved Across Calls |
|--------------------------|------------------------|
| Condition register       |                        |
| Bits 0-7 (CR0,CR1)       | no                     |
| Bits 8-22 (CR2,CR3,CR4)  | yes                    |
| Bits 23-31 (CR5,CR6,CR7) | no                     |
| Link register            | no                     |
| Count register           | no                     |
| XER register             | no                     |
| FPSCR register           | no                     |

## The Stack

The stack is a portion of storage that is used to hold local storage, register save areas, parameter lists, and call-chain data. The stack grows from higher addresses to lower addresses. A stack pointer register (register 1) is used to mark the current “top” of the stack.

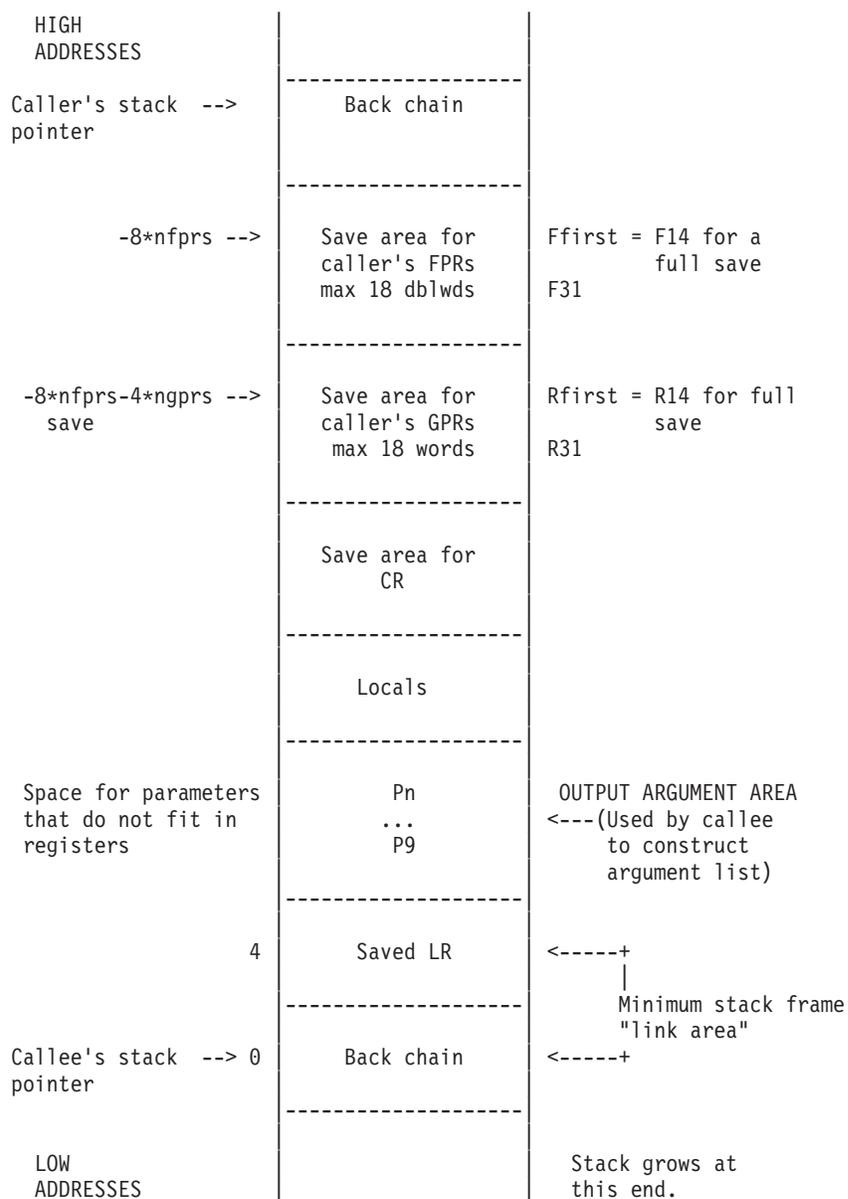
A stack frame is the portion of the stack that is used by a single procedure. The input parameters are considered part of the current stack frame. In a sense, each output argument belongs to both the caller’s and the callee’s stack frames. In either case, the stack frame size is best defined as the difference between the caller’s stack pointer and the callee’s.

The following diagrams show the storage maps of typical stack frames for 32-bit and 64-bit environments.

In these diagrams, the current routine has acquired a stack frame that allows it to call other functions. If the routine does not make any calls and there are no local variables or temporaries, and it does not need to save any non-volatile registers, the function need not allocate a stack frame. It can still use the register save area at the top of the caller’s stack frame, if needed.

The stack frame is double-word aligned.

### Run-time Stack for 32-bit Environment



### Run-time Stack for 64-bit Environment

|                                       |                                                  |                                                                                                |
|---------------------------------------|--------------------------------------------------|------------------------------------------------------------------------------------------------|
| Low Addresses                         |                                                  | Stack grows at this end.                                                                       |
| Callee's stack pointer --> 0          | Back chain                                       |                                                                                                |
| 8                                     | Saved CR                                         |                                                                                                |
| 16                                    | Saved LR                                         |                                                                                                |
| 24-32                                 | Reserved                                         | <--- LINK AREA (callee)                                                                        |
| 40                                    | Saved TOC                                        |                                                                                                |
| Space for P1-P8 is always reserved    | P1<br>...<br>Pn                                  | OUTPUT ARGUMENT AREA<br><---(Used by callee to construct argument list)                        |
|                                       | Callee's stack area                              | <--- LOCAL STACK AREA                                                                          |
|                                       |                                                  | (Possible word wasted for alignment.)                                                          |
| -8*ngprs --> save                     | Save area for caller's GPR<br>max 19 doublewords | Rfirst = R13 for full save<br>R31                                                              |
| -8*ngprs -->                          | Save area for caller's FPR<br>max 18 dblwds      | Ffirst = F14 for a full save<br>F31                                                            |
| Caller's stack pointer --> 0          | Back chain                                       |                                                                                                |
| 8                                     | Saved CR                                         |                                                                                                |
| 16                                    | Saved LR                                         |                                                                                                |
| 24-32                                 | Reserved                                         | <--- LINK AREA (caller)                                                                        |
| 40                                    | Saved TOC                                        |                                                                                                |
| Space for P1-P8 is always reserved 48 | P1<br>...<br>Pn                                  | INPUT PARAMETER AREA<br><---(Callee's input parameters found here. Is also caller's arg area.) |
|                                       | Caller's stack area                              |                                                                                                |
| High Addresses                        |                                                  |                                                                                                |

## The Link Area and Minimum Stack Frame

In a 32-bit environment, the link area consists of two words at offset zero from the callee's stack pointer on entry to a procedure. The first word contains the caller's back chain (pointer to the previous stack frame). The second word is the location where the caller saves the Link Register (LR), if it is needed.

In a 64-bit environment, this area consists of six doublewords at offset zero from the caller's stack pointer on entry to a procedure. The first doubleword contains the caller's back chain (stack pointer). The second doubleword is the location where the callee saves the Condition Register (CR) if it is needed. The third doubleword is the location where the callee's prolog code saves the Link Register if it is needed. The fourth doubleword is reserved for C **SETJMP** and **LONGJMP** processing, and the fifth doubleword is reserved for future use. The last doubleword (doubleword 6) is reserved for use by the global linkage routines that are used when calling routines in other object modules (for example, in shared libraries).

## The Input Parameter Area

In a 32-bit environment, the input parameters that do not fit in registers go into the output argument area (P9... Pn).

In a 64-bit environment, the input parameter area is a contiguous piece of storage reserved by the calling program to represent the register image of the input parameters of the callee. The input parameter area is double-word aligned and is located on the stack directly following the caller's link area. This area is at least 8 doublewords in size. If more than 8 doublewords of parameters are expected, they are stored as register images that start at positive offset 112 from the incoming stack pointer.

The first 8 doublewords only appear in registers at the call point, never in the stack. Remaining words are always in the stack, and they can also be in registers.

## The Register Save Area

In a 32-bit environment, the register save area provides the space that is needed to save all nonvolatile FPRs and GPRs used by the callee program. The FPRs are saved next to caller's minimum stack frame. The GPRs are saved below the FPRs (in lower addresses).

In a 64-bit environment, the register save area is double-word aligned. It provides the space that is needed to save all nonvolatile FPRs and GPRs used by the callee program. The FPRs are saved next to the link area. The GPRs are saved below the FPRs (in lower addresses). The called function may save the registers here even if it does not need to allocate a new stack frame. The system-defined stack floor includes the maximum possible save area:

32-bit platforms: 18\*8 for FPRs + 18\*4 for GPRs  
64-bit platforms: 18\*8 for FPRs + 19\*8 for GPRs

A callee needs only to save the nonvolatile registers that it actually uses.

## The Local Stack Area

The local stack area is the space that is allocated by the callee procedure for local variables and temporaries.

## The Output Parameter Area

In a 32-bit environment, the input parameters that do not fit in registers go into the output argument area (P9... Pn).

If more than 8 words are being passed, an extension list is constructed beginning at offset 8 from the current stack pointer.

The first 8 words only appear in registers at the call point, never in the stack. Remaining words are always in the stack, and they can also be in registers.

In a 64-bit environment, the output parameter area (P1...Pn) must be large enough to hold the largest parameter list of all procedures that the procedure that owns this stack frame calls. This area is at least 8 doublewords long, regardless of the length or existence of any argument list. If more than 8 doublewords are being passed, an extension list is constructed, which begins at offset 112 from the current stack pointer.

The first 8 doublewords only appear in registers at the call point, never in the stack. Remaining doublewords are always in the stack, and they can also be in registers.

---

## Linkage Convention for Argument Passing

The system linkage convention takes advantage of the large number of registers available. The linkage convention passes arguments in both GPRs and FPRs. Two fixed lists, R3-R10 and FP1-FP13, specify the GPRs and FPRs available for argument passing.

When there are more argument words than available argument GPRs and FPRs, the remaining words are passed in storage on the stack. The values in storage are the same as if they were in registers.

In a 64-bit environment, the size of the parameter area is sufficient to contain all the arguments passed on any call statement from a procedure that is associated with the stack frame. Although not all the arguments for a particular call actually appear in storage, it is convenient to consider them as forming a list in this area, each one occupying one or more words.

For call by reference (as is the default for Fortran), the address of the argument is passed in a register. The following information refers to call by value, as in C or as in Fortran when %VAL is used. For purposes of their appearance in the list, arguments are classified as floating-point values or non-floating-point values:

### In a 32-bit Environment

- Each **INTEGER(8)** and **LOGICAL(8)** argument requires two words.
- Any other non-floating-point scalar argument of intrinsic type or procedure/function pointers requires one word and appears in that word exactly as it would appear in a GPR. It is signed or unsigned/extended, if language semantics specify, and is word aligned.
- Each single-precision (**REAL(4)**) value occupies one word. Each double-precision (**REAL(8)**) value occupies two successive words in the list. Each extended-precision (**REAL(16)**) value occupies four successive words in the list.
- A **COMPLEX** value occupies twice as many words as a **REAL** value with the same kind type parameter.
- In Fortran and C, structure values are passed “val-by-ref”. That is, the compiler actually passes the address of a copy of the structure.

#### In a 64-bit Environment

- All non-floating-point values require one doubleword that is doubleword aligned.
- Each single-precision (**REAL(4)**) value and each double-precision (**REAL(8)**) value occupies one doubleword in the list. Each extended-precision (**REAL(16)**) value occupies two successive doublewords in the list.
- A **COMPLEX** value occupies twice as many doublewords as a **REAL** value with the same kind type parameter.
- In Fortran and C, structure values appear in successive words as they would anywhere in storage, satisfying all appropriate alignment requirements. Structures are aligned to a doubleword and occupy  $(\text{sizeof}(\text{struct } X)+7)/8$  doublewords, with any padding at the end. A structure that is smaller than a doubleword is left-justified within its doubleword or register. Larger structures can occupy multiple registers and may be passed partly in storage and partly in registers.
- Other aggregate values, including Pascal records, are passed “val-by-ref”. That is, the compiler actually passes their address and arranges for a copy to be made in the invoked program.
- A procedure or function pointer is passed as a pointer to the routine’s function descriptor; its first word contains its entry point address. (See “Pointers to Functions” on page 321 for more information.)

## Argument Passing Rules (by Value)

From the following illustration, we state these rules:

- In a 32-bit environment, arguments to called functions are passed in the GPRs and FPRs. Up to eight words are passed in GPR3-GPR10 and up to eight floating-point arguments in FPR1-FPR8. If fewer arguments are passed, unneeded registers are not loaded. If the passed arguments will not fit in registers, only enough space to hold the arguments that do not fit is allocated in the stack frame.
- In a 64-bit environment, if the called procedure treats the parameter list as a contiguous piece of storage (for example, if the address of a parameter is taken in C), the parameter registers are stored in the space reserved for them in the stack.
- A register image is stored on the stack.
- In a 64-bit environment, the argument area ( $P_1\dots P_n$ ) must be large enough to hold the largest parameter list.

Here is an example of a call to a function:

```
f(%val(11), %val(12), %val(13), %val(14), %val(15), %val(16), %val(17),
 %val(d1), %val(f1), %val(c1), %val(d2), %val(s1), %val(cx2))
```

where:

- l denotes integer(4) (fullword integer)
- d denotes real(8) (double precision)
- f denotes real(4) (real)
- s denotes integer(2) (halfword integer)
- c denotes character (one character)
- cx denotes complex(8) (double complex)

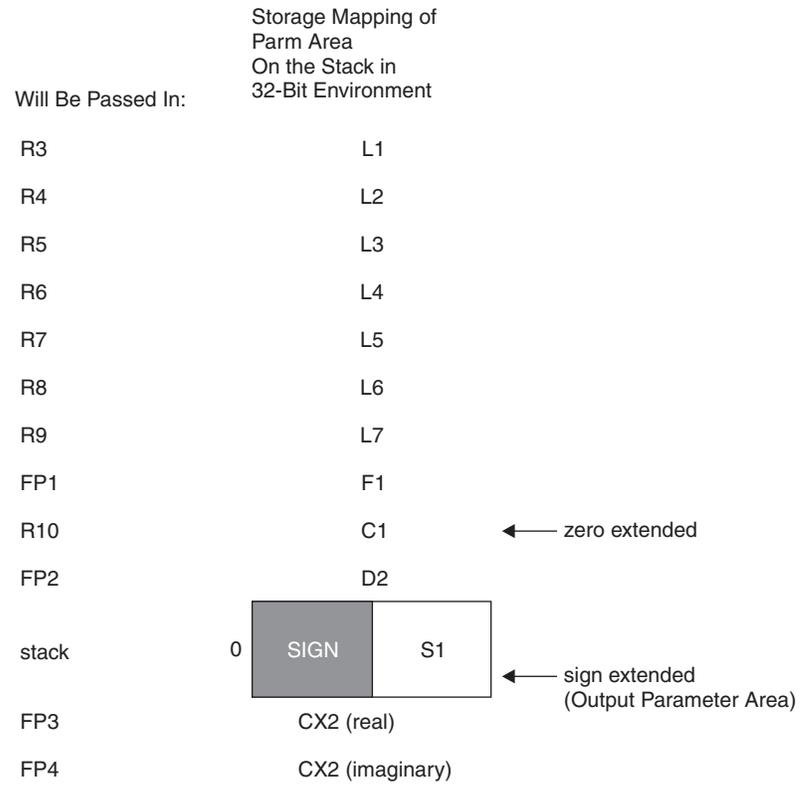


Figure 4. Storage Mapping of Parm Area On the Stack in 32-Bit Environment

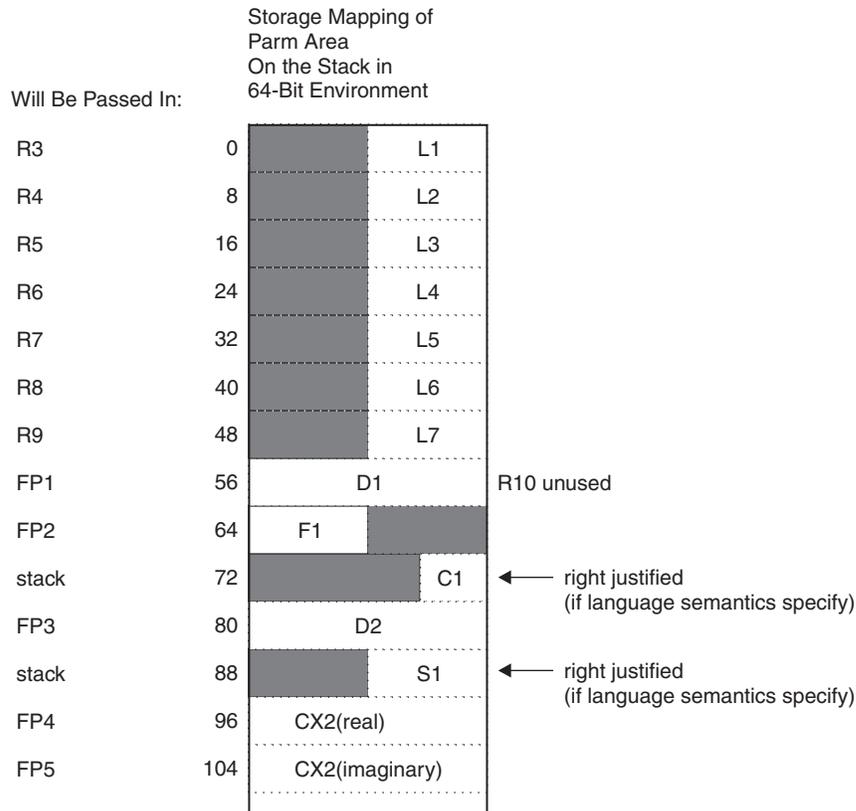


Figure 5. Storage Mapping of Parm Area On the Stack in 64-Bit Environment

## Order of Arguments in Argument List

The argument list is constructed in the following order. Items in the same bullet appear in the same order as in the procedure declaration, whether or not argument keywords are used in the call.

- All addresses or values (or both) of actual arguments <sup>4</sup>
- "Present" indicators for optional arguments that are passed by value
- Length arguments for strings <sup>4</sup>

---

## Linkage Convention for Function Calls

In 64-bit mode, a routine has two symbols associated with it: a function descriptor (*name*) and an entry point (*.name*). When a call is made to a routine, the program branches to the entry point directly. Excluding the loading of parameters (if any) in the proper registers, compilers expand calls to functions to the following two-instruction sequence:

```
BL .foo # Branch to foo
ORI R0,R0,0x0000 # Special NOP
```

The linker does one of two things when it encounters a **BL** instruction:

1. If *foo* is imported (not in the same object module), the linker changes the **BL** to *.foo* to a **BL** to *.glink* (global linkage routine) of *foo* and inserts the *.glink*

---

4. There may be other items in this list during Fortran-Fortran calls. However, they will not be visible to non-Fortran procedures that follow the calling rules in this section.

into the object module. Also, if a **NOP** instruction (`ORI R0,R0,0x0000`) immediately follows the **BL** instruction, the linker replaces the **NOP** instruction with the **LOAD** instruction `L R2, 20(R1)`.

2. If `foo` is bound in the same object module as its caller and a **LOAD** instruction `L R2,20(R1)` for 32-bit and `L R2,40(R1)` for 64-bit, or `ORI R0,R0,0` immediately follows the **BL** instruction, the linker replaces the **LOAD** instruction with a **NOP** (`ORI R0,R0,0`).

**Note:** For any export, the linker inserts the procedure's descriptor into the object module.

## Pointers to Functions

In 64-bit mode, a function pointer is a data type whose values range over procedure names. Variables of this type appear in several programming languages, such as C and Fortran. In Fortran, a dummy argument that appears in an **EXTERNAL** statement is a function pointer. Fortran provides support for the use of function pointers in contexts such as the target of a call statement or an actual argument of such a statement.

A function pointer is a fullword quantity that is the address of a function descriptor. The function descriptor is a 3-word object. The first word contains the address of the entry point of the procedure. The second has the address of the TOC of the object module in which the procedure is bound. The third is the environment pointer for some non-Fortran languages. There is only one function descriptor per entry point. It is bound into the same object module as the function it identifies if the function is external. The descriptor has an external name, which is the same as the function name but with a different storage class that uniquely identifies it. This descriptor name is used in all import or export operations.

## Function Values

Functions return their values according to type:

- In 32-bit mode, **INTEGER** and **LOGICAL** of kind 1, 2, and 4 are returned (sign/zero extended) in R3.
- In 64-bit mode, **INTEGER** and **LOGICAL** of kind 1, 2, and 4 are returned (right justified) in R3.
- In 64-bit mode, **INTEGER** and **LOGICAL** of kind 8 are returned in R3.
- **REAL** \*4 or \*8 are returned in FP1. **REAL** \*16 are returned in FP1 and FP2.
- **COMPLEX** \*8 or \*16 are returned in FP1 and FP2. **COMPLEX** \*32 are returned in FP1-FP4.
- In 32-bit mode when `-qfloat=complexgcc` is specified, **COMPLEX**\*8 is returned in R3-R4 and **COMPLEX**\*16 in R3-R6. In 64-bit mode, **COMPLEX**\*8 is returned in R3 and **COMPLEX**\*16 in R3-R4.
- Character strings are returned in a buffer allocated by the caller. The address and the length of this buffer are passed in R3 and R4 as hidden parameters. The first explicit parameter word is in R5, and all subsequent parameters are moved to the next word.
- Structures are returned in a buffer that is allocated by the caller. The address is passed in R3; there is no length. The first explicit parameter is in R4.

## The Stack Floor

In 64-bit mode, the stack floor is a system-defined address below which the stack cannot grow. All programs in the system must avoid accessing locations in the stack segment that are below the stack floor.

All programs must maintain other system invariants that are related to the stack:

- No data is saved or accessed from an address lower than the stack floor.
- The stack pointer is always valid. When the stack frame size is more than 32 767 bytes, you must take care to ensure that its value is changed in a single instruction. This step ensures that there is no timing window where a signal handler would either overlay the stack data or erroneously appear to overflow the stack segment.

## Stack Overflow

The linkage convention requires no explicit inline check for overflow. The operating system uses a storage protection mechanism to detect stores past the end of the stack segment.

---

## Prolog and Epilog

On entry to a procedure, you might have to do some or all of the following steps:

1. Save the link register.
2. If you use any of the CR bits 8-23 (CR2, CR3, CR4, CR5), save the CR.
3. Save any nonvolatile FPRs that are used by this procedure in the FPR save area.
4. Save all nonvolatile GPRs that are used by this procedure in the GPR save area.
5. Store back chain and decrement stack pointer by the size of the stack frame. Note that if a stack overflow occurs, it will be known immediately when the store of the back chain is done.

On exit from a procedure, you might have to perform some or all of the following steps:

1. Restore all GPRs saved.
2. Restore stack pointer to the value it had on entry.
3. Restore link register if necessary.
4. Restore bits 8-23 of the CR if necessary.
5. If you saved any FPRs, restore them.
6. Return to caller.

---

## Traceback

In 64-bit mode, the compiler supports the traceback mechanism, which symbolic debuggers need to unravel the call or return stack. Each object module has a traceback table in the text segment at the end of its code. This table contains information about the object module, including the type of object module, as well as stack frame and register information.

**Related Information:** You can make the traceback table smaller or remove it entirely with the “-qtbtable Option” on page 214.

---

## THREADLOCAL Common Blocks and ILC with C

Fortran **THREADLOCAL** common blocks are implemented using the thread-specific data facilities that are defined by the POSIX pthreads library. For additional information about thread-specific data areas, please refer to documentation on threads programming.

Internally, the storage for the thread-specific common block is allocated dynamically by the Fortran run-time library. The Fortran run-time library maintains a control structure that holds information about the common block. This control area is an external structure whose name is the name of the common block.

For example, if you declare a common block in Fortran as the following:

```
 common /myblock/ i
!ibm* threadlocal /myblock/
```

the Fortran compiler creates an external structure (or common area) that is named *myblock*, which contains control information about the thread-specific common block.

The control structure has the following layout and would be coded as such in C:

```
typedef struct {
 pthread_key_t key;
 int flags;
 void *unused_1;
 int unused_2;
} FORT_LOCAL_COMMON;
extern FORT_LOCAL_COMMON myblock;
```

The "key" field is a unique identifier that describes a threadlocal data area. Every threadlocal common block has its own key. The "flags" field indicates whether a key has been obtained for the common block. Within a C function, you should use the "key" in the control block in a call to **pthread\_getspecific** to obtain the thread-specific address of the threadlocal common area.

## Example

```
! Example 1: "fort_sub" is invoked by multiple threads. This is an invalid example
! because "fort_sub" and "another_sub" both declare /block/ to be THREADLOCAL.
! They intend to share the common block, but they are executed by different threads.
```

```
SUBROUTINE fort_sub()
COMMON /block/ j
INTEGER :: j
!IBM* THREADLOCAL /block/ ! Each thread executing fort_sub
 ! obtains its own copy of /block/.

INTEGER a(10)

...
!IBM* INDEPENDENT
DO index = 1,10
 CALL another_sub(a(i))
END DO
...

END SUBROUTINE fort_sub

SUBROUTINE another_sub(aa) ! Multiple threads are used to execute another_sub.
INTEGER aa
COMMON /block/ j ! Each thread obtains a new copy of the
 ! common block: /block/.
INTEGER :: j
!IBM* THREADLOCAL /block/
...
aa = j ! The value of 'j' is undefined.
END SUBROUTINE another_sub
```

For more information, see the **THREADLOCAL** directive, in the *XL Fortran Language Reference*.

---

## Problem Determination and Debugging

This section describes some methods you can use for locating and fixing problems in compiling or executing your programs.

---

### Understanding XL Fortran Error Messages

Most information about potential or actual problems comes through messages from the compiler or application program. These messages are written to the standard error output stream.

#### Error Severity

Compilation errors can have the following severity levels, which are displayed as part of some error messages:

- U** An unrecoverable error. Compilation failed because of an internal compiler error.
- S** A severe error. Compilation failed due to one of the following:
  - Conditions exist that the compiler could not correct. An object file is produced; however, you should not attempt to run the program.
  - An internal compiler table has overflowed. Processing of the program stops, and XL Fortran does not produce an object file.
  - An include file does not exist. Processing of the program stops, and XL Fortran does not produce an object file.
  - An unrecoverable program error has been detected. Processing of the source file stops, and XL Fortran does not produce an object file. You can usually correct this error by fixing any program errors that were reported during compilation.
- E** An error that the compiler can correct. The program should run correctly.
- W** Warning message. It does not signify an error but may indicate some unexpected condition.
- L** Warning message that was generated by one of the compiler options that check for conformance to various language levels. It may indicate a language feature that you should avoid if you are concerned about portability.
- I** Informational message. It does not indicate any error, just something that you should be aware of to avoid unexpected behavior.

#### Notes:

1. The message levels **S** and **U** indicate a compilation failure.
2. The message levels **I**, **L**, **W**, and **E** indicate that compilation was successful.

By default, the compiler stops without producing output files if it encounters a severe error (severity **S**). You can make the compiler stop for less severe errors by specifying a different severity with the **-qhalt** option. For example, with **-qhalt=e**, the compiler stops if it encounters any errors of severity **E** or higher severity. This technique can reduce the amount of compilation time that is needed to check the syntactic and semantic validity of a program. You can limit low-severity messages without stopping the compiler by using the **-qflag** option. If you simply want to prevent specific messages from going to the output stream, see “-qsuppress Option” on page 210.

## Compiler Return Code

The compiler return codes and their respective meanings are as follows:

|     |                                                                                                                                                                                                                                |
|-----|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 0   | The compiler did not encounter any errors severe enough to make it stop processing a compilation unit.                                                                                                                         |
| 1   | The compiler encountered an error of severity E or <i>halt_severity</i> (whichever is lower). Depending on the level of <i>halt_severity</i> , the compiler might have continued processing the compilation units with errors. |
| 40  | An option error.                                                                                                                                                                                                               |
| 41  | A configuration file error.                                                                                                                                                                                                    |
| 250 | An out-of-memory error. The compiler cannot allocate any more memory for its use.                                                                                                                                              |
| 251 | A signal received error. An unrecoverable error or interrupt signal is received.                                                                                                                                               |
| 252 | A file-not-found error.                                                                                                                                                                                                        |
| 253 | An input/output error. Cannot read or write files.                                                                                                                                                                             |
| 254 | A fork error. Cannot create a new process.                                                                                                                                                                                     |
| 255 | An error while executing a process.                                                                                                                                                                                            |

## Run-Time Return Code

If an XLF-compiled program ends abnormally, the return code to the operating system is 1.

If the program ends normally, the return code is 0 (by default) or is  $\text{MOD}(\text{digit\_string}, 256)$  if the program ends because of a **STOP** *digit\_string* statement.

## Understanding XL Fortran Messages

In addition to the diagnostic message issued, the source line and a pointer to the position in the source line at which the error was detected are printed or displayed if you specify the **-qsource** compiler option. If **-qnosource** is in effect, the file name, the line number, and the column position of the error are displayed with the message.

The format of an XL Fortran diagnostic message is:

```
▶▶—15—cc—--nnn— ———┌—————┐—————▶▶
 | (—severity_letter—) | message_text
```

where:

|       |                                                                     |
|-------|---------------------------------------------------------------------|
| 15    | Indicates an XL Fortran message                                     |
| cc    | Is the component number, as follows:                                |
| 00    | Indicates a code generation or optimization message                 |
| 01    | Indicates an XL Fortran common message                              |
| 11-20 | Indicates a Fortran-specific message                                |
| 25    | Indicates a run-time message from an XL Fortran application program |
| 85    | Indicates a loop-transformation message                             |
| 86    | Indicates an interprocedural analysis (IPA) message                 |
| nnn   | Is the message number                                               |

*severity\_letter* Indicates how serious the problem is, as described in the preceding section

*'message text'* Is the text describing the error

## Limiting the Number of Compile-Time Messages

If the compiler issues many low-severity (**I** or **W**) messages concerning problems you are aware of or do not care about, use the **-qflag** option or its short form **-w** to limit messages to high-severity ones:

```
E, S, and U messages go in listing; U messages are displayed on screen.
xlf95 -qflag=e:u program.f
```

```
E, S, and U messages go in listing and are displayed on screen.
```

```
xlf95 -w program.f
```

## Selecting the Language for Messages

By default, XL Fortran comes with messages in U.S. English only. You can also order translated message catalogs:

- Compiler messages in Japanese
- Run-time messages in Japanese

If compile-time messages are appearing in U.S. English when they should be in another language, verify that the correct message catalogs are installed and that the **LANG**, **LC\_MESSAGES**, and/or **LC\_ALL** environment variables are set accordingly.

If a run-time message appears in the wrong language, also ensure that your program calls the **setlocale** routine.

**Related Information:** See “Environment Variables for National Language Support” on page 12 and “Selecting the Language for Run-Time Messages” on page 34.

To determine which XL Fortran message catalogs are installed, use the following commands to list them:

```
rpm -ql xlf.cmp # compile-time messages
rpm -ql xlf.msg.rte # run-time messages
rpm -ql xlsmp.msg.rte # SMP run-time messages
```

The file names of the message catalogs are the same for all supported international languages, but they are placed in different directories.

**Note:** When you run an XL Fortran program on a system without the XL Fortran message catalogs, run-time error messages (mostly for I/O problems) are not displayed correctly; the program prints the message number but not the associated text. To prevent this problem, copy the XL Fortran message catalogs from **/opt/ibmcmp/msg** to a directory that is part of the **NLSPATH** environment-variable setting on the execution system.

---

## Fixing Installation or System Environment Problems

If individual users or all users on a particular machine have difficulty running the compiler, there may be a problem in the system environment. Here are some common problems and solutions:

---

xlf90: not found  
xlf90\_r: not found  
xlf95: not found  
xlf95\_r: not found  
xlf: not found  
xlf\_r: not found  
f77: not found  
fort77: not found  
f90: not found  
f95: not found

**Symptom:** The shell cannot locate the command to execute the compiler.

**Solution:** Make sure that your **PATH** environment variable includes the directory **/opt/ibmcomp/xlf/9.1/bin**. If the compiler is properly installed, the commands you need to execute it are in this directory.

---

**Could not load program *program***

**Error was: not enough space**

**Symptom:** The system cannot execute the compiler or an application program at all.

**Solution:** Set the storage limits for stack and data to “unlimited” for users who experience this problem. For example, you can set both your hard and soft limits with these **bash** commands:

```
ulimit -s unlimited
ulimit -d unlimited
```

Or, you may find it more convenient to edit the file **/etc/security/limits.conf** to give all users unlimited stack and data segments (by entering -1 for these fields).

If the storage problem is in an XLF-compiled program, using the **-qsave** or **-qsmallstack** option might prevent the program from exceeding the stack limit.

**Explanation:** The compiler allocates large internal data areas that may exceed the storage limits for a user. XLF-compiled programs place more data on the stack by default than in previous versions, also possibly exceeding the storage limit. Because it is difficult to determine precise values for the necessary limits, we recommend making them unlimited.

---

## Fixing Compile-Time Problems

The following sections discuss common problems you might encounter while compiling and how to avoid them.

**Could not load program *program***

**Could not load library *library\_name.so***

**Error was: no such file or directory**

**Solution:** Make sure the XL Fortran libraries are installed in **/opt/ibmcomp/xlf/9.1/lib** and **/opt/ibmcomp/xlf/9.1/lib64**, or set the **LD\_LIBRARY\_PATH** and **LD\_RUN\_PATH** environment variables to include the directory where **libxlf90.so** is installed if it is in a different directory. See “Setting Library Search Paths” on page 13 for details of this environment variable.

---

**Symptom:** Messages from the compiler or an XL Fortran application program are displayed in the wrong language.

**Solution:** Set the appropriate national language environment. You can set the national language for each user with the command **env**. Alternatively, each user can set one or more of the environment variables **LANG**, **NLS\_PATH**, **LC\_MESSAGES**, **LC\_TIME**, and **LC\_ALL**. If you are not familiar with the purposes of these variables, “Environment Variables for National Language Support” on page 12 provides details.

---

**Symptom:** A compilation fails with an I/O error.

**Solution:** Increase the size of the **/tmp** filesystem, or set the environment variable **TMPDIR** to the path of a filesystem that has more free space.

**Explanation:** The object file may have grown too large for the filesystem that holds it. The cause could be a very large compilation unit or initialization of all or part of a large array in a declaration.

---

**Symptom:** There are too many individual makefiles and compilation scripts to easily maintain or track.

**Solution:** Add stanzas to the configuration file, and create links to the compiler by using the names of these stanzas. By running the compiler with different command names, you can provide consistent groups of compiler options and other configuration settings to many users.

---

## Duplicating Extensions from Other Systems

Some ported programs may cause compilation problems because they rely on extensions that exist on other systems. XL Fortran supports many extensions like these, but some require compiler options to turn them on. See “Options for Compatibility” on page 62 for a list of these options and “Porting Programs to XL Fortran” on page 339 for a general discussion of porting.

## Isolating Problems with Individual Compilation Units

If you find that a particular compilation unit requires specific option settings to compile properly, you may find it more convenient to apply the settings in the source file through an `@PROCESS` directive. Depending on the arrangement of your files, this approach may be simpler than recompiling different files with different command-line options.

## Compiling with Thread-safe Commands

Thread-safe invocation commands, like `xl_f_r` or `xl_f90_r`, for example, use different search paths and call different modules than the non thread-safe invocations. Your programs should account for the different usages. Programs that compile and run successfully for one environment may produce unexpected results when compiled and run for a different use. The configuration file, `xl.f.cfg`, shows the paths, libraries, and so on for each invocation command. (See “Customizing the Configuration File” on page 14 for a sample configuration file and an explanation of its contents.)

## Running out of Machine Resources

If the operating system runs low on resources (page space or disk space) while one of the compiler components is running, you should receive one of the following messages:

```
1501-229 Compilation ended because of lack of space.
1517-011 Compilation ended. No more system resources available.
```

```
1501-053 (S) Too much initialized data.
1501-511. Compilation failed for file [filename].
```

You may need to increase the system page space and recompile your program. See the man page information `man 8 mkswap swapon` for more information about page space.

If your program produces a large object file, for example, by initializing all or part of a large array, you may need to do one of the following:

- Increase the size of the filesystem that holds the `/tmp` directory.
- Set the `TMPDIR` environment variable to a filesystem with a lot of free space.
- For very large arrays, initialize the array at run time rather than statically (at compile time).

---

## Fixing Link-Time Problems

After the XL Fortran compiler processes the source files, the linker links the resulting object files together. Any messages issued at this stage come from the `ld` command. A frequently encountered error and its solution are listed here for your convenience:

---

**filename.o: In function 'main':**

**filename.o(.text+0x14): undefined reference  
to 'p'**

**filename.o(.text+0x14): relocation truncated  
to fit: R\_PPC\_REL24 p**

**Symptom:** A program cannot be linked because of unresolved references.

**Explanation:** Either needed object files or libraries are not being used during linking, there is an error in the

specification of one or more external names, or there is an error in the specification of one or more procedure interfaces.

**Solution:** You may need to do one or more of the following actions:

- Compile again with the **-WI,-M** option to create a file that contains information about undefined symbols.
- Make sure that if you use the **-U** option, all intrinsic names are in lowercase.

---

## Fixing Run-Time Problems

XL Fortran issues error messages during the running of a program in either of the following cases:

- XL Fortran detects an input/output error. “Setting Run-Time Options” on page 35 explains how to control these kinds of messages.
- XL Fortran detects an exception error, and the default exception handler is installed (through the **-qsigtrap** option or a call to **SIGNAL**). To get a more descriptive message than Core dumped, you may need to run the program from within **gdb**.

The causes for run-time exceptions are listed in “XL Fortran Run-Time Exceptions” on page 49.

You can investigate errors that occur during the execution of a program by using a symbolic debugger, such as **gdb**.

## Duplicating Extensions from Other Systems

Some ported programs may not run correctly if they rely on extensions that are found on other systems. XL Fortran supports many such extensions, but you need to turn on compiler options to use some of them. See “Options for Compatibility” on page 62 for a list of these options and “Porting Programs to XL Fortran” on page 339 for a general discussion of porting.

## Mismatched Sizes or Types for Arguments

Arguments of different sizes or types might produce incorrect execution and results. To do the type-checking during the early stages of compilation, specify interface blocks for the procedures that are called within a program.

## Working around Problems when Optimizing

If you find that a program produces incorrect results when it is optimized and if you can isolate the problem to a particular variable, you might be able to work around the problem temporarily by declaring the variable as **VOLATILE**. This prevents some optimizations that affect the variable. (See **VOLATILE** in the *XL Fortran Language Reference*.) Because this is only a temporary solution, you should continue debugging your code until you resolve your problem, and then remove the **VOLATILE** keyword. If you are confident that the source code and program design are correct and you continue to have problems, contact your support organization to help resolve the problem.

## Input/Output Errors

If the error detected is an input/output error and you have specified **IOSTAT** on the input/output statement in error, the **IOSTAT** variable is assigned a value according to *Conditions and IOSTAT Values* in the *XL Fortran Language Reference*.

If you have installed the XL Fortran run-time message catalog on the system on which the program is executing, a message number and message text are issued to the terminal (standard error) for certain I/O errors. If you have specified **IOMSG** on the input/output statement, the **IOMSG** variable is assigned the error message text if an error is detected, or the content of **IOMSG** variable is not changed. If this catalog is not installed on the system, only the message number appears. Some of the settings in “Setting Run-Time Options” on page 35 allow you to turn some of these error messages on and off.

If a program fails while writing a large data file, you may need to increase the maximum file size limit for your user ID. You can do this through a shell command, such as **ulimit** in **bash**.

## Tracebacks and Core Dumps

If a run-time exception occurs and an appropriate exception handler is installed, a message and a traceback listing are displayed. Depending on the handler, a core file might be produced as well. You can then use a debugger to examine the location of the exception.

To produce a traceback listing without ending the program, call the **xl\_\_trbk** procedure:

```
IF (X .GT. Y) THEN ! X > Y indicates that something is wrong.
 PRINT *, 'Error - X should not be greater than Y'
 CALL XL__TRBK ! Generate a traceback listing.
 X = 0 ! The program continues.
END IF
```

See “Installing an Exception Handler” on page 258 for instructions about exception handlers and “XL Fortran Run-Time Exceptions” on page 49 for information about the causes of run-time exceptions.

---

## Debugging a Fortran 90 or Fortran 95 Program

For instructions on using your chosen debugger, consult the online help within the debugger or its documentation.

Always specify the **-g** option when compiling programs for debugging.

**Related information:** See “Options for Error Checking and Debugging” on page 58.



---

## Understanding XL Fortran Compiler Listings

Diagnostic information is placed in the output listing produced by the **-qlist**, **-qsource**, **-qxref**, **-qattr**, **-qreport**, and **-qlistopt** compiler options. The **-S** option generates an assembler listing in a separate file.

To locate the cause of a problem with the help of a listing, you can refer to the following:

- The source section (to see any compilation errors in the context of the source program)
- The attribute and cross-reference section (to find data objects that are misnamed or used without being declared or to find mismatched parameters)
- The transformation and object sections (to see if the generated code is similar to what you expect)

A heading identifies each major section of the listing. A string of greater than symbols precede the section heading so that you can easily locate its beginning:

```
>>>> section name
```

You can select which sections appear in the listing by specifying compiler options.

**Related Information:** See “Options That Control Listings and Messages” on page 61.

---

### Header Section

The listing file has a header section that contains the following items:

- A compiler identifier that consists of the following:
  - Compiler name
  - Version number
  - Release number
  - Modification number
  - Fix number
- Source file name
- Date of compilation
- Time of compilation

The header section is always present in a listing; it is the first line and appears only once. The following sections are repeated for each compilation unit when more than one compilation unit is present.

---

### Options Section

The options section is always present in a listing. There is a separate section for each compilation unit. It indicates the specified options that are in effect for the compilation unit. This information is useful when you have conflicting options. If you specify the **-qlistopt** compiler option, this section lists the settings for all options.

---

## Source Section

The source section contains the input source lines with a line number and, optionally, a file number. The file number indicates the source file (or include file) from which the source line originated. All main file source lines (those that are not from an include file) do not have the file number printed. Each include file has a file number associated with it, and source lines from include files have that file number printed. The file number appears on the left, the line number appears to its right, and the text of the source line is to the right of the line number. XL Fortran numbers lines relative to each file. The source lines and the numbers that are associated with them appear only if the **-qsource** compiler option is in effect. You can selectively print parts of the source by using the **@PROCESS** directives **SOURCE** and **NOSOURCE** throughout the program.

## Error Messages

If the **-qsource** option is in effect, the error messages are interspersed with the source listing. The error messages that are generated during the compilation process contain the following:

- The source line
- A line of indicators that point to the columns that are in error
- The error message, which consists of the following:
  - The 4-digit component number
  - The number of the error message
  - The severity level of the message
  - The text that describes the error

For example:

```
 2 | equivalence (i,j,i)
 a.
a - 1514-092: (E) Same name appears more than once in an
equivalence group.
```

If the **-qnosource** option is in effect, the error messages are all that appear in the source section, and an error message contains:

- The file name in quotation marks
- The line number and column position of the error
- The error message, which consists of the following:
  - The 4-digit component number
  - The number of the error message
  - The severity level of the message
  - The text that describes the error

For example:

```
"doc.f", line 6.11: 1513-039 (S) Number of arguments is not
permitted for INTRINSIC function abs.
```

---

## Transformation Report Section

If the **-qreport** option is in effect, a transformation report listing shows how XL Fortran optimized the program. This section displays pseudo-Fortran code that corresponds to the original source code, so that you can see parallelization and loop transformations that the **-qhot** and/or **-qsmp** options have generated.

### Sample Report

The following report was created for the program **t.f** using the `xlf -qhot -qreport t.f`

command.

#### Program t.f:

```
integer a(100, 100)
integer i,j

do i = 1 , 100
 do j = 1, 100
 a(i,j) = j
 end do
end do
end
```

#### Transformation Report:

>>>> SOURCE SECTION <<<<<

```
** _main === End of Compilation 1 ===
```

>>>> LOOP TRANSFORMATION SECTION <<<<<

```

 PROGRAM _main ()
4| IF (.FALSE.) GOTO lab_9
 @LoopIV0 = 0
 Id=1 DO @LoopIV0 = @LoopIV0, 99
5| IF (.FALSE.) GOTO lab_11
 @LoopIV1 = 0
 Id=2 DO @LoopIV1 = @LoopIV1, 99
 ! DIR_INDEPENDENT loopId = 0
6| a((@LoopIV1 + 1),(@LoopIV0 + 1)) = (@LoopIV0 + 1)
7| ENDDO
 lab_11
8| ENDDO
 lab_9
9| END PROGRAM _main
```

| Source File | Source Line | Loop Id | Action / Information                     |
|-------------|-------------|---------|------------------------------------------|
| -----       | -----       | -----   | -----                                    |
| 0           | 4           | 1       | Loop interchanging applied to loop nest. |

>>>> FILE TABLE SECTION <<<<<

---

## Attribute and Cross-Reference Section

This section provides information about the entities that are used in the compilation unit. It is present if the `-qxref` or `-qattr` compiler option is in effect. Depending on the options in effect, this section contains all or part of the following information about the entities that are used in the compilation unit:

- Names of the entities
- Attributes of the entities (if `-qattr` is in effect). Attribute information may include any or all of the following details:
  - The type
  - The class of the name
  - The relative address of the name
  - Alignment
  - Dimensions
  - For an array, whether it is allocatable
  - Whether it is a pointer, target, or integer pointer
  - Whether it is a parameter
  - Whether it is volatile
  - For a dummy argument, its intent, whether it is value, and whether it is optional
  - Private, public, protected, module
- Coordinates to indicate where you have defined, referenced, or modified the entities. If you declared the entity, the coordinates are marked with a \$. If you initialized the entity, the coordinates are marked with a \*. If you both declared and initialized the entity at the same place, the coordinates are marked with a &. If the entity is set, the coordinates are marked with a @. If the entity is referenced, the coordinates are not marked.

Class is one of the following:

- Automatic
- BSS (uninitialized static internal)
- Common
- Common block
- Construct name
- Controlled (for an allocatable object)
- Controlled automatic (for an automatic object)
- Defined assignment
- Defined operator
- Derived type definition
- Entry
- External subprogram
- Function
- Generic name
- Internal subprogram
- Intrinsic
- Module
- Module function
- Module subroutine
- Namelist
- Pointee
- Private component
- Program
- Reference parameter
- Renames
- Static
- Subroutine

- Use associated
- Value parameter

Type is one of the following:

- Byte
- Character
- Complex
- Derived type
- Integer
- Logical
- Real

If you specify the **full** suboption with **-qxref** or **-qattr**, XL Fortran reports all entities in the compilation unit. If you do not specify this suboption, only the entities you actually use appear.

---

## Object Section

XL Fortran produces this section only when the **-qlist** compiler option is in effect. It contains the object code listing, which shows the source line number, the instruction offset in hexadecimal notation, the assembler mnemonic of the instruction, and the hexadecimal value of the instruction. On the right side, it also shows the cycle time of the instruction and the intermediate language of the compiler. Finally, the total cycle time (straight-line execution time) and the total number of machine instructions that are produced are displayed. There is a separate section for each compilation unit.

---

## File Table Section

This section contains a table that shows the file number and file name for each main source file and include file used. It also lists the line number of the main source file at which the include file is referenced. This section is always present.

---

## Compilation Unit Epilogue Section

This is the last section of the listing for each compilation unit. It contains the diagnostics summary and indicates whether the unit was compiled successfully. This section is not present in the listing if the file contains only one compilation unit.

---

## Compilation Epilogue Section

The compilation epilogue section occurs only once at the end of the listing. At completion of the compilation, XL Fortran presents a summary of the compilation: number of source records that were read, compilation start time, compilation end time, total compilation time, total CPU time, and virtual CPU time. This section is always present in a listing.

**Related Information:** Sample programs are shown in Appendix A, "Sample Fortran Programs," on page 345.



---

## Porting Programs to XL Fortran

XL Fortran provides many features intended to make it easier to take programs that were originally written for other computer systems or compilers and recompile them with XL Fortran.

---

### Outline of the Porting Process

The process for porting a typical program looks like this:

1. Identify any nonportable language extensions or subroutines that you used in the original program. Check to see which of these XL Fortran supports:
  - Language extensions are identified in the *XL Fortran Language Reference*.
  - Some extensions require you to specify an XL Fortran compiler option; you can find these options listed in Table 7 on page 63.
2. For any nonportable features that XL Fortran does not support, modify the source files to remove or work around them.
3. Do the same for any implementation-dependent features. For example, if your program relies on the representation of floating-point values or uses system-specific file names, you may need to change it.
4. Compile the program with XL Fortran. If any compilation problems occur, fix them and recompile and fix any additional errors until the program compiles successfully.
5. Run the XLF-compiled program and compare the output with the output from the other system. If the results are substantially different, there are probably still some implementation-specific features that need to be changed. If the results are only marginally different (for example, if XL Fortran produces a different number of digits of precision or a number differs in the last decimal place), decide whether the difference is significant enough to investigate further. You may be able to fix these differences.

Before porting programs to XL Fortran, read the tips in the following sections so that you know in advance what compatibility features XL Fortran offers.

---

### Portability of Directives

XL Fortran supports many directives available with other Fortran products. This ensures easy portability between products. If your code contains *trigger\_constants* other than the defaults in XL Fortran, you can use the **-qdirective** compiler option to specify them. For instance, if you are porting CRAY code contained in a file *xx.f*, you would use the following command to add the CRAY *trigger\_constant*:

```
xlf95 xx.f -qdirective=mic\$\
```

For fixed source form code, in addition to the ! value for the *trigger\_head* portion of the directive, XL Fortran also supports the *trigger\_head* values **C**, **c**, and **\***.

For more information, see “-qdirective Option” on page 120.

XL Fortran supports a number of programming terms as synonyms to ease the effort of porting code from other Fortran products. Those terms that are supported are dependent on context, as indicated in the following tables:

Table 25. PARALLEL DO Clauses and Their XL Fortran Synonyms

| PARALLEL DO Clause       | XL Fortran Synonym |
|--------------------------|--------------------|
| LASTLOCAL                | LASTPRIVATE        |
| LOCAL                    | PRIVATE            |
| MP_SCHETYPE<br>and CHUNK | SCHEDULE           |
| SAVELAST                 | LASTPRIVATE        |
| SHARE                    | SHARED             |
| NEW                      | PRIVATE            |

Table 26. PARALLEL DO Scheduling Types and Their XL Fortran Synonyms

| Scheduling Type | XL Fortran Synonym |
|-----------------|--------------------|
| GSS             | GUIDED             |
| INTERLEAVE      | STATIC(1)          |
| INTERLEAVED     | STATIC(1)          |
| INTERLEAVE(n)   | STATIC(n)          |
| INTERLEAVED(n)  | STATIC(n)          |
| SIMPLE          | STATIC             |

Table 27. PARALLEL SECTIONS Clauses and Their XL Fortran Synonyms

| PARALLEL SECTIONS Clause | XL Fortran Synonym |
|--------------------------|--------------------|
| LOCAL                    | PRIVATE            |
| SHARE                    | SHARED             |
| NEW                      | PRIVATE            |

## NEW

Use the **NEW** directive to specify which variables should be local in a **PARALLEL DO** loop or a **PARALLEL SECTIONS** construct. This directive performs the same function as the **PRIVATE** clause of the **PARALLEL DO** directive and **PARALLEL SECTIONS** directive.

### Background Information

The **NEW** directive only takes effect if you specify the **-qsmp** compiler option.

### Syntax

►►—NEW—*named\_variable\_list*—◄◄

The **NEW** directive must immediately follow either a **PARALLEL DO** directive or a **PARALLEL SECTIONS** directive.

If you specify the **NEW** directive, you must specify the corresponding **PARALLEL DO** or **PARALLEL SECTIONS** directive with no clauses.

If the **NEW** directive follows the **PARALLEL DO** directive, the first noncomment line (not including other directives) following the **NEW** directive must be a **DO** loop. This line cannot be an infinite **DO** or **DO WHILE** loop.

A variable name in the *named\_variable\_list* of the **NEW** directive has the same restrictions as a variable name appearing in the **PRIVATE** clause of the **PARALLEL DO** directive or a **PRIVATE** clause of the **PARALLEL SECTIONS** directive. See the sections on the **PARALLEL DO** directive and the **PARALLEL SECTIONS** construct in the *XL Fortran Language Reference*.

### Examples

```
INTEGER A(10), C(10)
REAL B(10)
INTEGER FUNC(100)
!SMP$ PARALLEL DO
!SMP$ NEW I, TMP
 DO I = 1, 10
 TMP = A(I) + COS(B(I))
 C(I) = TMP + FUNC(I)
 END DO
```

---

## Common Industry Extensions That XL Fortran Supports

XL Fortran allows many of the same FORTRAN 77 extensions as other popular compilers, including:

| <b>Extension</b>                                                                                                                                                                                                                                                                                                                          | <b>Refer to XL Fortran Language Reference Section(s)</b> |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------|
| Typeless constants                                                                                                                                                                                                                                                                                                                        | <i>Typeless Literal Constants</i>                        |
| <i>*len</i> length specifiers for types                                                                                                                                                                                                                                                                                                   | <i>The Data Types</i>                                    |
| <b>BYTE</b> data type                                                                                                                                                                                                                                                                                                                     | <b>BYTE</b>                                              |
| Long variable names                                                                                                                                                                                                                                                                                                                       | <i>Names</i>                                             |
| Lower case                                                                                                                                                                                                                                                                                                                                | <i>Names</i>                                             |
| Mixing integers and logicals (with <b>-qintlog</b> option)                                                                                                                                                                                                                                                                                | <i>Evaluation of Expressions</i>                         |
| Character-count <b>Q</b> edit descriptor (with <b>-qqcount</b> option)                                                                                                                                                                                                                                                                    | <i>Q (Character Count) Editing</i>                       |
| Intrinsics for counting set bits in registers and determining data-object parity                                                                                                                                                                                                                                                          | <b>POPCNT, POPPAR</b>                                    |
| 64-bit data types ( <b>INTEGER(8)</b> , <b>REAL(8)</b> , <b>COMPLEX(8)</b> , and <b>LOGICAL(8)</b> ), including support for default 64-bit types (with <b>-qintsize</b> and <b>-qrealsize</b> options)                                                                                                                                    | <i>Integer Real Complex Logical</i>                      |
| Integer <b>POINTERS</b> , similar to those supported by CRAY and Sun compilers. (XL Fortran integer pointer arithmetic uses increments of one byte, while the increment on CRAY computers is eight bytes. You may need to multiply pointer increments and decrements by eight to make programs ported from CRAY computers work properly.) | <b>POINTER(integer)</b>                                  |
| Conditional vector merge (CVMGx) intrinsic functions                                                                                                                                                                                                                                                                                      | <i>CVMGx (TSOURCE, FSOURCE, MASK)</i>                    |
| Date and time service and utility functions ( <b>rtc</b> , <b>irtc</b> , <b>jdate</b> , <b>clock_</b> , <b>timef</b> , and <b>date</b> )                                                                                                                                                                                                  | <i>Service and Utility Procedures</i>                    |
| <b>STRUCTURE</b> , <b>UNION</b> , and <b>MAP</b> constructs                                                                                                                                                                                                                                                                               | <i>Structure Components, Union and Map</i>               |

### Mixing Data Types in Statements

The **-qctyp1ss** option lets you use character constant expressions in the same places that you use typeless constants. The **-qintlog** option lets you use integer expressions where you can use logicals, and vice versa. A kind type parameter must not be replaced with a logical constant even if **-qintlog** is on, nor by a character constant even if **-qctyp1ss** is on, nor can it be a typeless constant.

### Date and Time Routines

Date and time routines, such as **dtime**, **etime**, and **jdate**, are accessible as Fortran subroutines.

### Other libc Routines

A number of other popular routines from the **libc** library, such as **flush**, **getenv**, and **system**, are also accessible as Fortran subroutines.

## Changing the Default Sizes of Data Types

For porting from machines with larger or smaller word sizes, the `-qintsize` option lets you specify the default size for integers and logicals. The `-qrealsize` option lets you specify the default size for reals and complex components.

## Name Conflicts between Your Procedures and XL Fortran Intrinsic Procedures

If you have procedures with the same names as any XL Fortran intrinsic procedures, the program calls the intrinsic procedure. (This situation is more likely with the addition of the many new Fortran 90 and Fortran 95 intrinsic procedures.)

If you still want to call your procedure, add explicit interfaces, **EXTERNAL** statements, or **PROCEDURE** statements for any procedures with conflicting names, or use the `-qextern` option when compiling.

## Reproducing Results from Other Systems

XL Fortran provides settings through the `-qfloat` option that help make floating-point results consistent with those from other IEEE systems; this subject is discussed in “Duplicating the Floating-Point Results of Other Systems” on page 257.

## Finding Nonstandard Extensions

XL Fortran supports a number of extensions to various language standards. Many of these extensions are so common that you need to keep in mind, when you port programs to other systems, that not all compilers have them. To find such extensions in your XL Fortran programs before beginning a porting effort, use the `-qlanglvl` option:

```
$ # -qnoobject stops the compiler after parsing all the source,
$ # giving a fast way to check for errors.
$ # Look for anything above the base F77 standard.
$ xlf -qnoobject -qlanglvl=77std f77prog.f
...
$ # Look for anything above the F90 standard.
$ xlf90 -qnoobject -qlanglvl=90std use_in_2000.f
...
$ # Look for anything above the F95 standard.
$ xlf95 -qnoobject -qlanglvl=95std use_in_2000.f
...
```

**Related Information:** See “`-qlanglvl` Option” on page 156 and “`-qport` Option” on page 182.



---

## Appendix A. Sample Fortran Programs

The following programs are provided as coding examples for XL Fortran. A number of these samples illustrate various aspects of SMP programming that may be new to many users. If you are new to SMP programming, you should examine these samples to gain a better understanding of the SMP coding style. Every attempt has been made to internally document key areas of the source to assist you in this effort.

You can compile and execute the first program to verify that the compiler is installed correctly and your user ID is set up to execute Fortran programs.

---

### Example 1 - XL Fortran Source File

```
PROGRAM CALCULATE
!
! Program to calculate the sum of up to n values of x**3
! where negative values are ignored.
!
 IMPLICIT NONE
 INTEGER I,N
 REAL SUM,X,Y
 READ(*,*) N
 SUM=0
 DO I=1,N
 READ(*,*) X
 IF (X.GE.0) THEN
 Y=X**3
 SUM=SUM+Y
 END IF
 END DO
 WRITE(*,*) 'This is the sum of the positive cubes:',SUM
END
```

### Execution Results

Here is what happens when you run the program:

```
$ a.out
5
37
22
-4
19
6
This is the sum of the positive cubes: 68376.00000
```

---

## Example 2 - Valid C Routine Source File

```
/*
 * *****
 * This is a main function that creates threads to execute the Fortran
 * test subroutines.
 * *****
 */
#include <pthread.h>
#include <stdio.h>
#include <errno.h>

extern char *optarg;
extern int optind;

static char *prog_name;

#define MAX_NUM_THREADS 100

void *f_mt_exec(void *);
void f_pre_mt_exec(void);
void f_post_mt_exec(int *);

void
usage(void)
{
 fprintf(stderr, "Usage: %s -t number_of_threads.\n", prog_name);
 exit(-1);
}

main(int argc, char *argv[])
{
 int i, c, rc;
 int num_of_threads, n[MAX_NUM_THREADS];
 char *num_of_threads_p;
 pthread_attr_t attr;
 pthread_t tid[MAX_NUM_THREADS];

 prog_name = argv[0];
 while ((c = getopt(argc, argv, "t")) != EOF)
 {
 switch (c)
 {
 case 't':
 break;

 default:
 usage();
 break;
 }
 }

 argc -= optind;
 argv += optind;
 if (argc < 1)
 {
 usage();
 }

 num_of_threads_p = argv[0];
 if ((num_of_threads = atoi(num_of_threads_p)) == 0)
 {
 fprintf(stderr,
 "%s: Invalid number of threads to be created <%s>\n", prog_name,
 num_of_threads_p);
 exit(1);
 }
}
```

```

}
else if (num_of_threads > MAX_NUM_THREADS)
{
 fprintf(stderr,
 "%s: Cannot create more than 100 threads.\n", prog_name);
 exit(1);
}
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

/* *****
 * Execute the Fortran subroutine that prepares for multi-threaded
 * execution.
 * *****
 */
f_pre_mt_exec();

for (i = 0; i < num_of_threads; i++)
{
 n[i] = i;
 rc = pthread_create(&tid[i], &attr, f_mt_exec, (void *)&n[i]);
 if (rc != 0)
 {
 fprintf(stderr, "Failed to create thread %d.\n", i);

 exit(1);
 }
}
/* The attribute is no longer needed after threads are created. */
pthread_attr_destroy(&attr);
for (i = 0; i < num_of_threads; i++)
{
 rc = pthread_join(tid[i], NULL);
 if (rc != 0)
 {
 fprintf(stderr, "Failed to join thread %d. \n", i);
 }
}
/*
 * Execute the Fortran subroutine that does the check after
 * multi-threaded execution.
 */
f_post_mt_exec(&num_of_threads);

exit(0);
}

```

```

! *****
! This test case tests the writing list-directed to a single external
! file by many threads.
! *****

```

```

subroutine f_pre_mt_exec()
integer array(1000)
common /x/ array

do i = 1, 1000
 array(i) = i
end do

open(10, file="fun10.out", form="formatted", status="replace")
end

subroutine f_post_mt_exec(number_of_threads)
integer array(1000), array1(1000)
common /x/ array

```

```

close(10)
open(10, file="fun10.out", form="formatted")
do j = 1, number_of_threads
 read(10, *) array1

 do i = 1, 1000
 if (array1(i) /= array(i)) then
 print *, "Result is wrong."
 stop
 endif
 end do
end do
close(10, status="delete")
print *, "Normal ending."
end

subroutine f_mt_exec(thread_number)
integer thread_number
integer array(1000)
common /x/ array

write(10, *) array
end

```

---

### Example 3 - Valid Fortran SMP Source File

```

!*****
!* This example uses a PARALLEL construct and a DO construct *
!* to calculate the value of pi. *
!*****
program compute_pi
integer n, i
real*8 w, x, pi, f, a
f(a) = 4.d0 / (1.d0 + a*a) !! function to integrate

pi = 0.0d0
!$OMP PARALLEL private(x, w, n), shared(pi)
n = 10000 !! number of intervals
w = 1.0d0/n !! calculate the interval size
!$OMP DO reduction(+: pi)
do i = 1, n
 x = w * (i - 0.5d0)
 pi = pi + f(x)
enddo
!$OMP END DO
!$OMP END PARALLEL
print *, "Computed pi = ", pi
end

```

---

### Example 4 - Invalid Fortran SMP Source File

```

!*****
!* In this example, fort_sub is invoked by multiple threads. *
!* * *
!* This example is not valid because *
!* fort_sub and another_sub both declare /block/ to be *
!* THREADLOCAL. They intend to share the common block, but *
!* they are executed via different threads. *
!* * *
!* To "fix" this problem, one of the following approaches can *
!* be taken: *
!* (1) The code for another_sub should be brought into the loop.*
!* (2) "j" should be passed as an argument to another_sub, and *
!* the declaration for /block/ should be removed from *
!* another_sub. *

```

```

!* (3) The loop should be marked as "do not parallelize" by *
!* using the directive "!SMP$ PARALLEL DO IF(.FALSE.)". *
!*****
subroutine fort_sub()

 common /block/ j
 integer :: j
 !IBM* THREADLOCAL /block/ ! Each thread executing fort_sub
 ! obtains its own copy of /block/.

 integer a(10)

 ...
 !IBM* INDEPENDENT
 do index = 1,10
 call another_sub(a(i))
 enddo
 ...

end subroutine fort_sub

subroutine another_sub(aa) ! Multiple threads are used to
 integer aa ! execute another_sub.
 common /block/ j ! Each thread obtains a new copy
 integer :: j ! of the common block /block/.
 !IBM* THREADLOCAL /block/

 aa = j ! The value of "j" is undefined.
end subroutine another_sub

```

---

## Programming Examples Using the Pthreads Library Module

```

!*****
!* Example 5 : Create a thread with Round_Robin scheduling policy.*
!* For simplicity, we do not show any codes for error checking, *
!* which would be necessary in a real program. *
!*****
 use, intrinsic::f_pthread
 integer(4) ret_val
 type(f_pthread_attr_t) attr
 type(f_pthread_t) thr

 ret_val = f_pthread_attr_init(attr)
 ret_val = f_pthread_attr_setschedpolicy(attr, SCHED_RR)
 ret_val = f_pthread_attr_setinheritsched(attr, PTHREAD_EXPLICIT_SCHED)
 ret_val = f_pthread_create(thr, attr, FLAG_DEFAULT, ent, integer_arg)
 ret_val = f_pthread_attr_destroy(attr)


```

Before you can manipulate a pthread attribute object, you need to create and initialize it. The appropriate interfaces must be called to manipulate the attribute objects. A call to `f_pthread_attr_setschedpolicy` sets the scheduling policy attribute to Round\_Robin. Note that this does not affect newly created threads that inherit the scheduling property from the creating thread. For these threads, we explicitly call `f_pthread_attr_setinheritsched` to override the default inheritance attribute. The rest of the code is self-explanatory.

```

!*****
!* Example 6 : Thread safety *
!* In this example, we show that thread safety can be achieved *
!* by using the push-pop cleanup stack for each thread. We *
!* assume that the thread is in deferred cancellability-enabled *
!* state. This means that any thread-cancel requests will be *
!* put on hold until a cancellation point is encountered. *
!* Note that f_pthread_cond_wait provides a *
!* cancellation point. *
!*****

```

```

!*****
use, intrinsic::f_thread
integer(4) ret_val
type(f_thread_mutex_t) mutex
type(f_thread_cond_t) cond
pointer(p, byte)
! Initialize mutex and condition variables before using them.
! For global variables this should be done in a module, so that they
! can be used by all threads. If they are local, other threads
! will not see them. Furthermore, they must be managed carefully
! (for example, destroy them before returning, to avoid dangling and
! undefined objects).
mutex = PTHREAD_MUTEX_INITIALIZER
cond = PTHREAD_COND_INITIALIZER

.....
! Doing something

.....

! This thread needs to allocate some memory area used to
! synchronize with other threads. However, when it waits on a
! condition variable, this thread may be canceled by another
! thread. The allocated memory may be lost if no measures are
! taken in advance. This will cause memory leakage.

ret_val = f_thread_mutex_lock(mutex)
p = malloc(%val(4096))

! Check condition. If it is not true, wait for it.
! This should be a loop.

! Since memory has been allocated, cleanup must be registered
! for safety during condition waiting.

ret_val = f_thread_cleanup_push(mycleanup, FLAG_DEFAULT, p)
ret_val = f_thread_cond_wait(cond, mutex)

! If this thread returns from condition waiting, the cleanup
! should be de-registered.

call f_thread_cleanup_pop(0) ! not execute
ret_val = f_thread_mutex_unlock(mutex)

! This thread will take care of p for the rest of its life.
.....

! mycleanup looks like:

subroutine mycleanup(passed_in)
 pointer(passed_in, byte)
 external free

 call free(%val(passed_in))
end subroutine mycleanup

```

---

## Appendix B. XL Fortran Technical Information

This section contains details about XL Fortran that advanced programmers may need to diagnose unusual problems, run the compiler in a specialized environment, or do other things that a casual programmer is rarely concerned with.

---

### The Compiler Phases

The typical compiler invocation command executes some or all of the following programs in sequence. For link-time optimizations, some of the phases will be executed more than once during a compilation. As each program runs, the results are sent to the next step in the sequence.

1. A preprocessor
2. The compiler, which consists of the following phases:
  - a. Front-end parsing and semantic analysis
  - b. Loop transformations
  - c. Interprocedural analysis
  - d. Optimization
  - e. Register allocation
  - f. Final assembly
3. The assembler (for any `.s` files)
4. The linker `ld`

---

### External Names in XL Fortran Libraries

To minimize naming conflicts between user-defined names and the names that are defined in the run-time libraries, the names of input/output routines in the run-time libraries are prefixed with an underscore(`_`), or `_xl`.

---

### The XL Fortran Run-Time Environment

Object code that the XL Fortran compiler produces often invokes compiler-supplied subprograms at run time to handle certain complex tasks. These subprograms are collected into several libraries.

The function of the XL Fortran Run-Time Environment may be divided into these main categories:

- Support for Fortran I/O operations
- Mathematical calculation
- Operating-system services
- Support for SMP parallelization

The XL Fortran Run-Time Environment also produces run-time diagnostic messages in the national language appropriate for your system. Unless you bind statically, you cannot run object code produced by the XL Fortran compiler without the XL Fortran Run-Time Environment.

The XL Fortran Run-Time Environment is upward-compatible. Programs that are compiled with a given level of the run-time environment and a given level of the operating system require the same or higher levels of both the run-time environment and the operating system to run.

## External Names in the Run-Time Environment

Run-time subprograms are collected into libraries. By default, the compiler invocation command also invokes the linker and gives it the names of the libraries that contain run-time subprograms called by Fortran object code.

The names of these run-time subprograms are external symbols. When object code that is produced by the XL Fortran compiler calls a run-time subprogram, the `.o` object code file contains an external symbol reference to the name of the subprogram. A library contains an external symbol definition for the subprogram. The linker resolves the run-time subprogram call with the subprogram definition.

You should avoid using names in your XL Fortran program that conflict with names of run-time subprograms. Conflict can arise under two conditions:

- The name of a subroutine, function, or common block that is defined in a Fortran program has the same name as a library subprogram.
- The Fortran program calls a subroutine or function with the same name as a library subprogram but does not supply a definition for the called subroutine or function.

---

### Technical Details of the `-qfloat=hsflt` Option

The `-qfloat=hsflt` option is unsafe for optimized programs that compute floating-point values that are outside the range of representation of single precision, not just outside the range of the result type. The range of representation includes both the precision and the exponent range.

Even when you follow the rules that are stated in the preceding paragraph and in “`-qfloat` Option” on page 133, programs that are sensitive to precision differences might not produce expected results. Because `-qfloat=hsflt` is not compliant with IEEE, programs will not always run as expected.

For example, in the following program, `X.EQ.Y` may be true or may be false:

```
REAL X, Y, A(2)
DOUBLE PRECISION Z
LOGICAL SAME

READ *, Z
X = Z
Y = Z
IF (X.EQ.Y) SAME = .TRUE.
! ...
! ... Calculations that do not change X or Y
! ...
CALL SUB(X) ! X is stored in memory with truncated fraction.
IF (X.EQ.Y) THEN ! Result might be different than before.
...

A(1) = Z
X = Z
A(2) = 1. ! A(1) is stored in memory with truncated fraction.
IF (A(1).EQ.X) THEN ! Result might be different than expected.
...

```

If the value of `Z` has fractional bits that are outside the precision of a single-precision variable, these bits may be preserved in some cases and lost in others. This makes the exact results unpredictable when the double-precision value

| of Z is assigned to single-precision variables. For example, passing the variable as a  
| dummy argument causes its value to be stored in memory with a fraction that is  
| truncated rather than rounded.

---

## Implementation Details for `-qautodbl` Promotion and Padding

The following sections provide additional details about how the `-qautodbl` option works, to allow you to predict what happens during promotion and padding.

### Terminology

The *storage relationship* between two data objects determines the relative starting addresses and the relative sizes of the objects. The `-qautodbl` option tries to preserve this relationship as much as possible.

Data objects can also have a *value relationship*, which determines how changes to one object affect another. For example, a program might store a value into one variable, and then read the value through a different storage-associated variable. With `-qautodbl` in effect, the representation of one or both variables might be different, so the value relationship is not always preserved.

An object that is affected by this option may be:

- *Promoted*, meaning that it is converted to a higher-precision data type. Usually, the resulting object is twice as large as it would be by default. Promotion applies to constants, variables, derived-type components, arrays, and functions (which include intrinsic functions) of the appropriate types.

**Note:** `BYTE`, `INTEGER`, `LOGICAL`, and `CHARACTER` objects are never promoted.

- *Padded*, meaning that the object keeps its original type but is followed by undefined storage space. Padding applies to `BYTE`, `INTEGER`, `LOGICAL`, and nonpromoted `REAL` and `COMPLEX` objects that may share storage space with promoted items. For safety, `POINTERS`, `TARGETS`, actual and dummy arguments, members of `COMMON` blocks, structures, pointee arrays, and pointee `COMPLEX` objects are always padded appropriately depending on the `-qautodbl` suboption. This is true whether or not they share storage with promoted objects.

Space added for padding ensures that the storage-sharing relationship that existed before conversion is maintained. For example, if array elements `I(20)` and `R(10)` start at the same address by default and if the elements of `R` are promoted and become twice as large, the elements of `I` are padded so that `I(20)` and `R(10)` still start at the same address.

Except for unformatted I/O statements, which read and write any padding that is present within structures, I/O statements do not process padding.

**Note:** The compiler does not pad `CHARACTER` objects.

## Examples of Storage Relationships for `-qautodbl` Suboptions

The examples in this section illustrate storage-sharing relationships between the following types of entities:

- **REAL(4)**
- **REAL(8)**
- **REAL(16)**
- **COMPLEX(4)**
- **COMPLEX(8)**
- **COMPLEX(16)**
- **INTEGER(8)**
- **INTEGER(4)**
- **CHARACTER(16)**.

**Note:** In the diagrams, solid lines represent the actual data, and dashed lines represent padding.

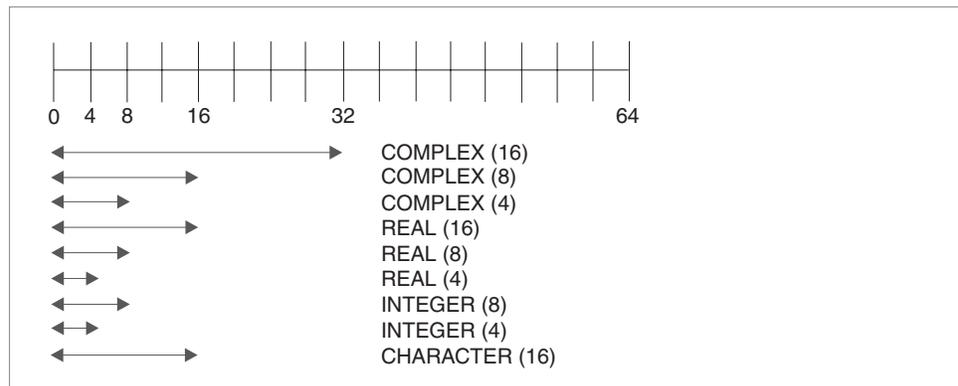


Figure 6. Storage Relationships without the `-qautodbl` Option

The figure above illustrates the default storage-sharing relationship of the compiler.

```
@process autodbl(none)
 block data
 complex(4) x8 /(1.123456789e0,2.123456789e0)/
 real(16) r16(2) /1.123q0,2.123q0/
 integer(8) i8(2) /1000,2000/
 character*5 c(2) /"abcde","12345"/
 common /named/ x8,r16,i8,c
 end

 subroutine s()
 complex(4) x8
 real(16) r16(2)
 integer(8) i8(2)
 character*5 c(2)
 common /named/ x8,r16,i8,c
 ! x8 = (1.123456e0,2.123456e0) ! promotion did not occur
 ! r16(1) = 1.123q0 ! no padding
 ! r16(2) = 2.123q0 ! no padding
 ! i8(1) = 1000 ! no padding
 ! i8(2) = 2000 ! no padding
 ! c(1) = "abcde" ! no padding
 ! c(2) = "12345" ! no padding
 end subroutine s
```

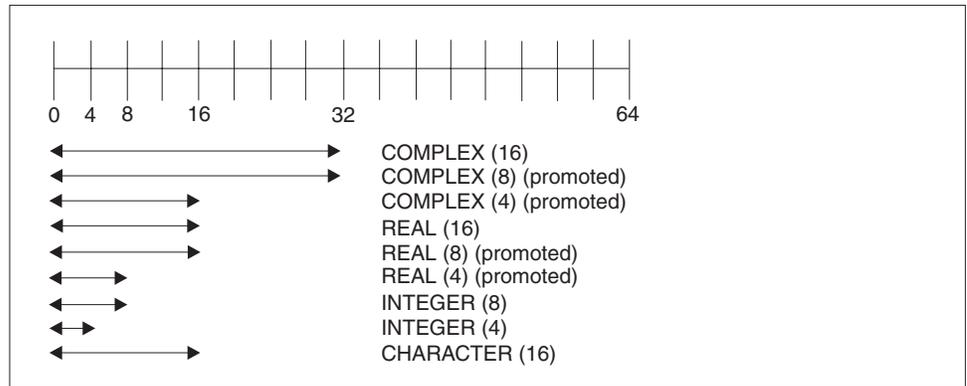


Figure 7. Storage Relationships with `-qautodbl=dbl`

```

@process autodbl(dbl)
 block data
 complex(4) x8
 real(16) r16(2) /1.123q0,2.123q0/
 real(8) r8
 real(4) r4 /1.123456789e0/
 integer(8) i8(2) /1000,2000/
 character*5 c(2) /"abcde","12345"/
 equivalence (x8,r8)
 common /named/ r16,i8,c,r4
 ! Storage relationship between r8 and x8 is preserved.
 ! Data values are NOT preserved between r8 and x8.
 end

 subroutine s()
 real(16) r16(2)
 real(8) r4
 integer(8) i8(2)
 character*5 c(2)
 common /named/ r16,i8,c,r4
 ! r16(1) = 1.123q0 ! no padding
 ! r16(2) = 2.123q0 ! no padding
 ! r4 = 1.123456789d0 ! promotion occurred
 ! i8(1) = 1000 ! no padding
 ! i8(2) = 2000 ! no padding
 ! c(1) = "abcde" ! no padding
 ! c(2) = "12345" ! no padding
 end subroutine s

```

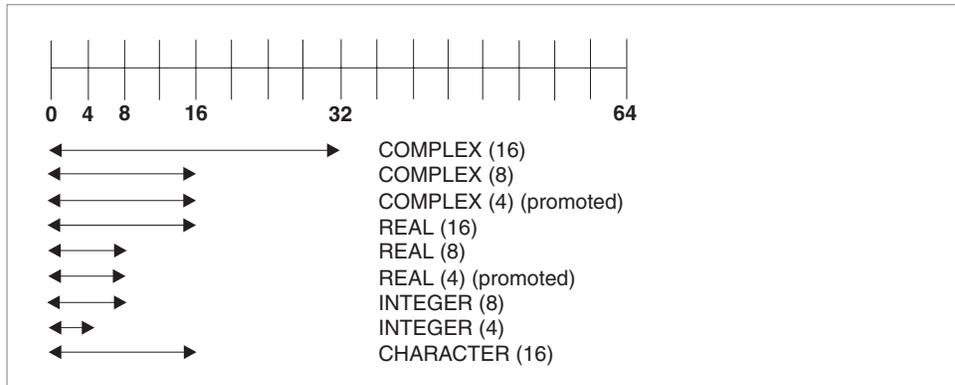


Figure 8. Storage Relationships with `-qautobl=dbl4`

```
@process autodb1(db14)
 complex(8) x16 /(1.123456789d0,2.123456789d0)/
 complex(4) x8
 real(4) r4(2)
 equivalence (x16,x8,r4)
! Storage relationship between r4 and x8 is preserved.
! Data values between r4 and x8 are preserved.
! x16 = (1.123456789d0,2.123456789d0) ! promotion did not occur
! x8 = (1.123456789d0,2.123456789d0) ! promotion occurred
! r4(1) = 1.123456789d0 ! promotion occurred
! r4(2) = 2.123456789d0 ! promotion occurred
end
```

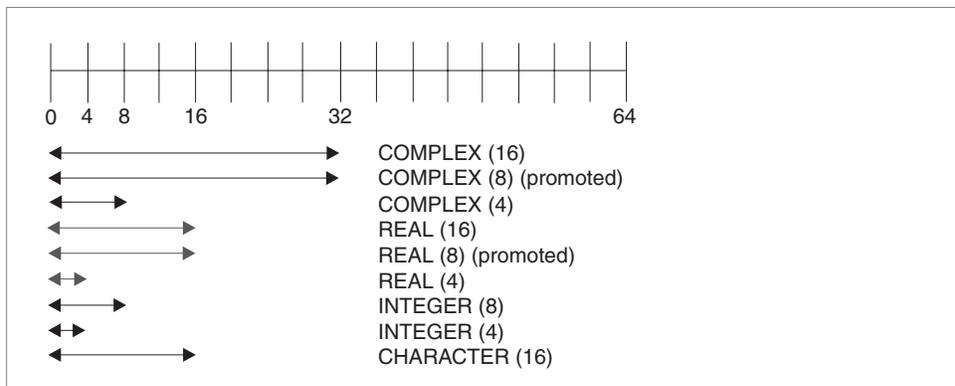


Figure 9. Storage Relationships with `-qautodbl=dbl8`

```
@process autodb1(db18)
 complex(8) x16 /(1.123456789123456789d0,2.123456789123456789d0)/
 complex(4) x8
 real(8) r8(2)
 equivalence (x16,x8,r8)
! Storage relationship between r8 and x16 is preserved.
! Data values between r8 and x16 are preserved.
! x16 = (1.123456789123456789q0,2.123456789123456789q0)
!
! x8 = upper 8 bytes of r8(1) ! promotion did not occur
! r8(1) = 1.123456789123456789q0 ! promotion occurred
! r8(2) = 2.123456789123456789q0 ! promotion occurred
end
```

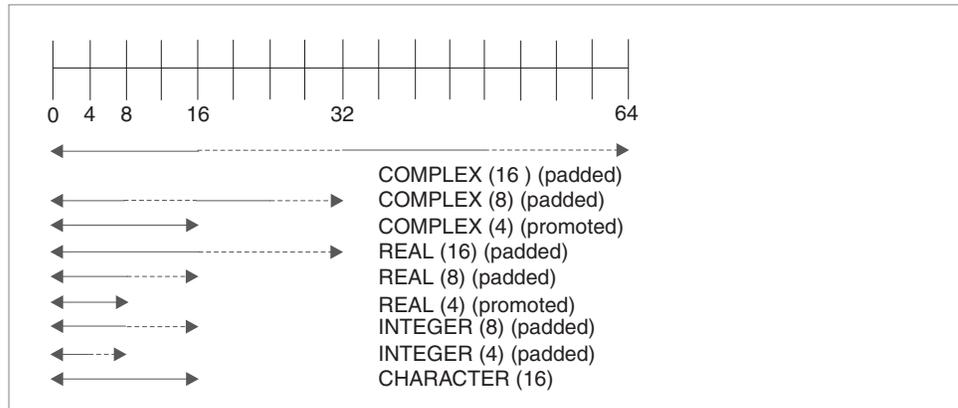


Figure 10. Storage Relationships with *-qautodbl=dblpad4*

In the figure above, the dashed lines represent the padding.

```
@process autodbl(dblpad4)
 complex(8) x16 /(1.123456789d0,2.123456789d0)/
 complex(4) x8
 real(4) r4(2)
 integer(8) i8(2)
 equivalence(x16,x8,r4,i8)
! Storage relationship among all entities is preserved.
! Date values between x8 and r4 are preserved.
! x16 = (1.123456789d0,2.123456789d0) ! padding occurred
! x8 = (upper 8 bytes of x16, 8 byte pad) ! promotion occurred
! r4(1) = real(x8) ! promotion occurred
! r4(2) = imag(x8) ! promotion occurred
! i8(1) = real(x16) ! padding occurred
! i8(2) = imag(x16) ! padding occurred
end
```

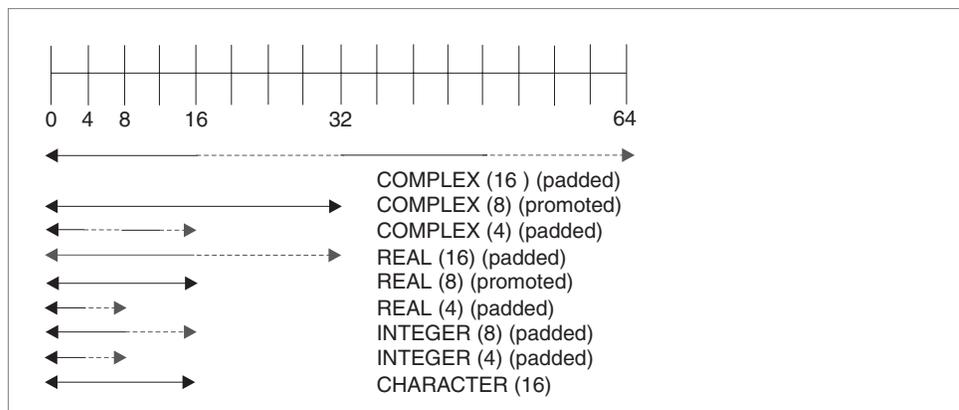


Figure 11. Storage Relationships with *-qautodbl=dblpad8*

In the figure above, the dashed lines represent the padding.

```
@process autodbl(dblpad8)
 complex(8) x16 /(1.123456789123456789d0,2.123456789123456789d0)/
 complex(4) x8
 real(8) r8(2)
 integer(8) i8(2)
 byte b(16)
 equivalence(x16,x8,r8,i8,b)
! Storage relationship among all entities is preserved.
```

```

! Data values between r8 and x16 are preserved.
! Data values between i8 and b are preserved.
! x16 = (1.123456789123456789q0,2.123456789123456789q0)
!
! ! promotion occurred
! x8 = upper 8 bytes of r8(1) ! padding occurred
! r8(1) = real(x16) ! promotion occurred
! r8(2) = imag(x16) ! promotion occurred
! i8(1) = upper 8 bytes of real(x16) ! padding occurred
! i8(2) = upper 8 bytes of imag(x16) ! padding occurred
! b(1:8)= i8(1) ! padding occurred
! b(9:16)= i8(2) ! padding occurred
end

```

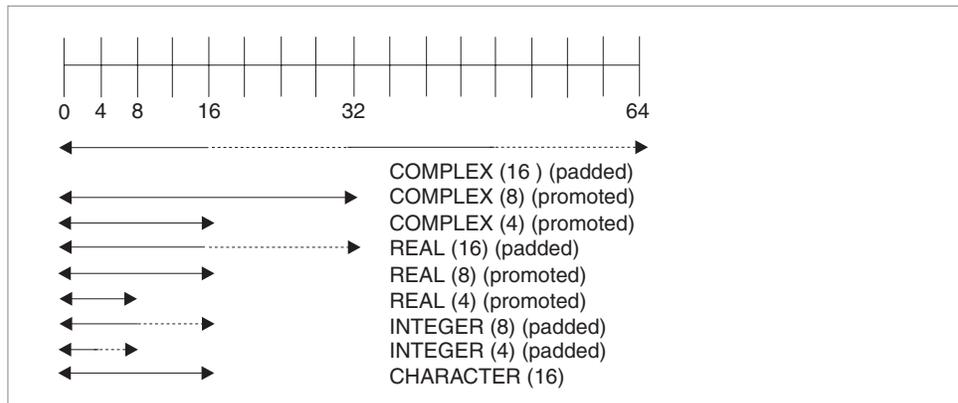


Figure 12. Storage Relationships with `-qautodbl=dblpad`

In the figure above, the dashed lines represent the padding.

```

@process autodbl(db1pad)
 block data
 complex(4) x8 /(1.123456789e0,2.123456789e0)/
 real(16) r16(2) /1.123q0,2.123q0/
 integer(8) i8(2) /1000,2000/
 character*5 c(2) /"abcde","12345"/
 common /named/ x8,r16,i8,c
 end
 subroutine s()
 complex(8) x8
 real(16) r16(4)
 integer(8) i8(4)
 character*5 c(2)
 common /named/ x8,r16,i8,c
! x8 = (1.123456789d0,2.123456789d0) ! promotion occurred
! r16(1) = 1.123q0 ! padding occurred
! r16(3) = 2.123q0 ! padding occurred
! i8(1) = 1000 ! padding occurred
! i8(3) = 2000 ! padding occurred
! c(1) = "abcde" ! no padding occurred
! c(2) = "12345" ! no padding occurred
 end subroutine s

```

---

## Appendix C. Using the Mathematical Acceleration Subsystem (MASS)

XL Fortran Advanced Edition V9.1 for Linux ships the Mathematical Acceleration Subsystem (MASS), a set of libraries of tuned mathematical intrinsic functions that provide improved performance over the corresponding standard math library functions. The accuracy and exception handling might not be identical in MASS functions and standard math library functions.

The MASS libraries on Linux consist of a library of vector functions, described in “Using the Vector Libraries.” “Compiling and Linking a Program with MASS” on page 360 describes how to compile and link a program that uses the MASS libraries.

---

### Using the Vector Libraries

The MASS libraries for Linux are a subset of the MASS libraries for AIX. On Linux, 32-bit and 64-bit objects may not be mixed in a single library, so two versions of the MASS library are provided: **libmassvp4.a** (32-bit) and **libmassvp4\_64.a** (64-bit).

The single-precision and double-precision functions contained in the vector libraries are summarized in Table 28. To provide the interface declarations for the functions, include **massv.include** in your source files.

Table 28. MASS Vector Library Functions

| Double-precision function | Single-precision function | Arguments | Description                                                          |
|---------------------------|---------------------------|-----------|----------------------------------------------------------------------|
| vrec                      | vsrec                     | (y,x,n)   | Sets y(i) to the reciprocal of x(i), for i=1,..,n                    |
| vrsqrt                    | vsrsqrt                   | (y,x,n)   | Sets y(i) to the reciprocal of the square root of x(i), for i=1,..,n |
| vsqrt                     | vssqrt                    | (y,x,n)   | Sets y(i) to the square root of x(i), for i=1,..,n                   |

The following example shows interface declarations for some of the MASS double-precision vector functions:

```
interface
 subroutine vsqrt (y, x, n)
 real*8 y(*), x(*)
 integer n ! Sets y(i) to the square root of x(i), for i=1,..,n
 end subroutine vsqrt

 subroutine vrsqrt (y, x, n)
 real*8 y(*), x(*)
 integer n ! Sets y(i) to the reciprocal of the square root of x(i),
 ! for i=1,..,n
 end subroutine vrsqrt
end interface
```

The following example shows interface declarations for some of the MASS single-precision vector functions:

```

interface

subroutine vssqrt (y, x, n)
 real*4 y(*), x(*)
 integer n ! Sets y(i) to the square root of x(i), for i=1,..,n
end subroutine vssqrt

subroutine vsrsqrt (y, x, n)
 real*4 y(*), x(*)
 integer n ! Sets y(i) to the reciprocal of the square root of x(i),
 ! for i=1,..,n
end subroutine vsrsqrt

end interface

```

## Consistency of MASS Vector Functions

All of the functions in the MASS vector libraries are consistent, in the sense that a given input value will always produce the same result, regardless of its position in the vector, and regardless of the vector length.

---

## Compiling and Linking a Program with MASS

To compile an application that calls the routines in the MASS libraries, specify **massvp4** (32-bit) or **massvp4\_64** (64-bit) on the **-l** linker option. For example, if the MASS libraries are installed in the default directory, you could specify one of the following:

```

xlf progf.f -o progf -lmassvp4
xlf progf.f -o progf -lmassvp4_64 -q64

```

The MASS functions must run in the round-to-nearest rounding mode and with floating-point exception trapping disabled. (These are the default compilation settings.)

## Appendix D. XL Fortran Internal Limits

| Language Feature                                                                                                                                     | Limit                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Maximum number of iterations performed by <b>DO</b> loops with loop control with index variable of type INTEGER( <i>n</i> ) for <i>n</i> = 1, 2 or 4 | $(2^{**31})-1$                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| Maximum number of iterations performed by <b>DO</b> loops with loop control with index variable of type INTEGER(8)                                   | $(2^{**63})-1$                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| Maximum character format field width                                                                                                                 | $(2^{**31})-1$                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| Maximum length of a format specification                                                                                                             | $(2^{**31})-1$                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| Maximum length of Hollerith and character constant edit descriptors                                                                                  | $(2^{**31})-1$                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| Maximum length of a fixed source form statement                                                                                                      | 6 700                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| Maximum length of a free source form statement                                                                                                       | 6 700                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| Maximum number of continuation lines                                                                                                                 | n/a <b>1</b>                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| Maximum number of nested <b>INCLUDE</b> lines                                                                                                        | 64                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| Maximum number of nested interface blocks                                                                                                            | 1 024                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| Maximum number of statement numbers in a computed <b>GOTO</b>                                                                                        | 999                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| Maximum number of times a format code can be repeated                                                                                                | $(2^{**31})-1$                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| Allowable record numbers and record lengths for input/output files in 32-bit mode                                                                    | The record number can be up to $(2^{**63})-1$ . The maximum record length is $(2^{**31})-1$ bytes.                                                                                                                                                                                                                                                                                                                                                 |
| Allowable record numbers and record lengths for input/output files in 64-bit mode                                                                    | The record number can be up to $(2^{**63})-1$ , and the record length can be up to $(2^{**63})-1$ bytes.<br><br>However, for unformatted sequential files, you must use the <b>uwidth=64</b> run-time option for the record length to be greater than $(2^{**31})-1$ and up to $(2^{**63})-1$ . If you use the default <b>uwidth=32</b> run-time option, the maximum length of a record in an unformatted sequential file is $(2^{**31})-1$ bytes. |
| Allowable bound range of an array dimension                                                                                                          | The bound of an array dimension can be positive, negative, or zero within the range $-(2^{**31})$ to $2^{**31}-1$ in 32-bit mode, or $-(2^{**63})$ to $2^{**63}-1$ in 64-bit mode.                                                                                                                                                                                                                                                                 |
| Allowable external unit numbers                                                                                                                      | 0 to $(2^{**31})-1$ <b>2</b>                                                                                                                                                                                                                                                                                                                                                                                                                       |
| Maximum numeric format field width                                                                                                                   | 2 000                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| Maximum number of concurrent open files                                                                                                              | 1 024 <b>3</b>                                                                                                                                                                                                                                                                                                                                                                                                                                     |

**1** You can have as many continuation lines as you need to create a statement with a maximum of 6 700 bytes.

- 2** The value must be representable in an `INTEGER(4)` object, even if specified by an `INTEGER(8)` variable.
- 3** In practice, this value is somewhat lower because of files that the run-time system may open, such as the preconnected units 0, 5, and 6.

---

## Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing  
IBM Corporation  
North Castle Drive  
Armonk, NY 10504-1785  
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation  
Licensing  
2-31 Roppongi 3-chome, Minato-ku  
Tokyo 106, Japan

**The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:**

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation  
Lab Director  
IBM Canada Limited  
8200 Warden Avenue  
Markham, Ontario, Canada  
L6G 1C7

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.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

#### COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

This software and documentation are based in part on the Fourth Berkeley Software Distribution under license from the Regents of the University of California. We acknowledge the following institution for its role in this product's development: the Electrical Engineering and Computer Sciences Department at the Berkeley campus.

OpenMP is a trademark of the OpenMP Architecture Review Board. Portions of this document may have been derived from the *OpenMP Fortran Language Application Program Interface, Version 2.0* (November 2000) specification. Copyright 1997-2000 OpenMP Architecture Review Board.

---

## Programming Interface Information

Programming interface information is intended to help you create application software using this program.

General-use programming interfaces allow the customer to write application software that obtain the services of this program's tools.

However, this information may also contain diagnosis, modification, and tuning information. Diagnosis, modification, and tuning information is provided to help you debug your application software.

**Note:** Do not use this diagnosis, modification, and tuning information as a programming interface because it is subject to change.

---

## Trademarks and Service Marks

The following terms, used in this publication, are trademarks or service marks of the International Business Machines Corporation in the United States or other countries or both:

|                              |            |         |
|------------------------------|------------|---------|
| IBM                          | IBM (logo) | POWER3  |
| POWER4                       | POWER5     | PowerPC |
| PowerPC Architecture<br>z/OS | pSeries    | SAA     |

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

UNIX is a registered trademark of the Open Group in the United States and other countries.

Windows is a trademark of Microsoft Corporation in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.



---

## Glossary

This glossary defines terms that are commonly used in this document. It includes definitions developed by the American National Standards Institute (ANSI) and entries from the *IBM Dictionary of Computing*.

### A

**active processor.** See *online processor*.

**actual argument.** An expression, variable, procedure, or alternate return specifier that is specified in a procedure reference.

**alias.** A single piece of storage that can be accessed through more than a single name. Each name is an alias for that storage.

**alphabetic character.** A letter or other symbol, excluding digits, used in a language. Usually the uppercase and lowercase letters A through Z plus other special symbols (such as \$ and \_) allowed by a particular language.

**alphanumeric.** Pertaining to a character set that contains letters, digits, and usually other characters, such as punctuation marks and mathematical symbols.

**American National Standard Code for Information Interchange.** See *ASCII*.

**argument.** An expression that is passed to a function or subroutine. See also *actual argument*, *dummy argument*.

**argument association.** The relationship between an actual argument and a dummy argument during the invoking of a procedure.

**arithmetic constant.** A constant of type integer, real, or complex.

**arithmetic expression.** One or more arithmetic operators and arithmetic primaries, the evaluation of which produces a numeric value. An arithmetic expression can be an unsigned arithmetic constant, the name of an arithmetic constant, or a reference to an arithmetic variable, function reference, or a combination of such primaries formed by using arithmetic operators and parentheses.

**arithmetic operator.** A symbol that directs the performance of an arithmetic operation. The intrinsic arithmetic operators are:

+ addition  
- subtraction  
\* multiplication  
/ division  
\*\* exponentiation

**array.** An entity that contains an ordered group of scalar data. All objects in an array have the same data type and type parameters.

**array declarator.** The part of a statement that describes an array used in a program unit. It indicates the name of the array, the number of dimensions it contains, and the size of each dimension.

**array element.** A single data item in an array, identified by the array name and one or more subscripts. See also *subscript*.

**array name.** The name of an ordered set of data items.

**array section.** A subobject that is an array and is not a structure component.

**ASCII.** The standard code, using a coded character set consisting of 7-bit coded characters (8-bits including parity check), that is used for information interchange among data processing systems, data communication systems, and associated equipment. The ASCII set consists of control characters and graphic characters. See also *Unicode*.

**asynchronous.** Pertaining to events that are not synchronized in time or do not occur in regular or predictable time intervals. For example, input events are controlled by the user; the program can read them later.

**assignment statement.** An executable statement that defines or redefines a variable based on the result of expression evaluation.

**associate name.** The name by which a selector of an ASSOCIATE construct is known within the construct.

**attribute.** A property of a data object that may be specified in a type declaration statement, attribute specification statement, or through a default setting.

**automatic parallelization.** The process by which the compiler attempts to parallelize both explicitly coded DO loops and DO loops generated by the compiler for array language.

### B

**binary constant.** A constant that is made of one or more binary digits (0 and 1).

**bind.** To relate an identifier to another object in a program; for example, to relate an identifier to a value, an address or another identifier, or to associate formal parameters and actual parameters.

**blank common.** An unnamed common block.

**block data subprogram.** A subprogram headed by a **BLOCK DATA** statement and used to initialize variables in named common blocks.

**bss storage.** Uninitialized static storage.

**busy-wait.** The state in which a thread keeps executing in a tight loop looking for more work once it has completed all of its work and there is no new work to do.

**byte constant.** A named constant that is of type byte.

**byte type.** A data type representing a one-byte storage area that can be used wherever a **LOGICAL(1)**, **CHARACTER(1)**, or **INTEGER(1)** can be used.

## C

**character constant.** A string of one or more alphabetic characters enclosed in apostrophes or double quotation marks.

**character expression.** A character object, a character-valued function reference, or a sequence of them separated by the concatenation operator, with optional parentheses.

**character operator.** A symbol that represents an operation, such as concatenation (`//`), to be performed on character data.

**character set.** All the valid characters for a programming language or for a computer system.

**character string.** A sequence of consecutive characters.

**character substring.** A contiguous portion of a character string.

**character type.** A data type that consists of alphanumeric characters. See also *data type*.

**chunk.** A subset of consecutive loop iterations.

**collating sequence.** The sequence in which the characters are ordered for the purpose of sorting, merging, comparing, and processing indexed data sequentially.

**comment.** A language construct for the inclusion of text in a program that has no effect on the execution of the program.

**common block.** A storage area that may be referred to by a calling program and one or more subprograms.

**compile.** To translate a source program into an executable program (an object program).

**compiler directive.** Source code that controls what XL Fortran does rather than what the user program does.

**complex constant.** An ordered pair of real or integer constants separated by a comma and enclosed in parentheses. The first constant of the pair is the real part of the complex number; the second is the imaginary part.

**complex number.** A number consisting of an ordered pair of real numbers, expressible in the form **a+bi**, where **a** and **b** are real numbers and **i** squared equals -1.

**complex type.** A data type that represents the values of complex numbers. The value is expressed as an ordered pair of real data items separated by a comma and enclosed in parentheses. The first item represents the real part of the complex number; the second represents the imaginary part.

**conform.** To adhere to a prevailing standard. An executable program conforms to the Fortran 95 Standard if it uses only those forms and relationships described therein and if the executable program has an interpretation according to the Fortran 95 Standard. A program unit conforms to the Fortran 95 Standard if it can be included in an executable program in a manner that allows the executable program to be standard-conforming. A processor conforms to the standard if it executes standard-conforming programs in a manner that fulfills the interpretations prescribed in the standard.

**connected unit.** In XL Fortran, a unit that is connected to a file in one of three ways: explicitly via the **OPEN** statement to a named file, implicitly, or by preconnection.

**constant.** A data object with a value that does not change. The four classes of constants specify numbers (arithmetic), truth values (logical), character data (character), and typeless data (hexadecimal, octal, and binary). See also *variable*.

**construct.** A sequence of statements starting with a **SELECT CASE**, **DO**, **IF**, or **WHERE** statement and ending with the corresponding terminal statement.

**continuation line.** A line that continues a statement beyond its initial line.

**control statement.** A statement that is used to alter the continuous sequential invocation of statements; a control statement may be a conditional statement, such as **IF**, or an imperative statement, such as **STOP**.

## D

**data object.** A variable, constant, or subobject of a constant.

**data striping.** Spreading data across multiple storage devices so that I/O operations can be performed in parallel for better performance. Also known as *disk striping*.

**data transfer statement.** A **READ**, **WRITE**, or **PRINT** statement.

**data type.** The properties and internal representation that characterize data and functions. The intrinsic types are integer, real, complex, logical, and character. See also *intrinsic*.

**debug line.** Allowed only for fixed source form, a line containing source code that is to be used for debugging. Debug lines are defined by a D or X in column 1. The handling of debug lines is controlled by the **-qdlines** and **-qxlines** compiler options.

**default initialization.** The initialization of an object with a value specified as part of a derived type definition.

**definable variable.** A variable whose value can be changed by the appearance of its name or designator on the left of an assignment statement.

**delimiters.** A pair of parentheses or slashes (or both) used to enclose syntactic lists.

**denormalized number.** An IEEE number with a very small absolute value and lowered precision. A denormalized number is represented by a zero exponent and a non-zero fraction.

**derived type.** A type whose data have components, each of which is either of intrinsic type or of another derived type.

**digit.** A character that represents a nonnegative integer. For example, any of the numerals from 0 through 9.

**directive.** A type of comment that provides instructions and information to the compiler.

**disk striping.** See *data striping*.

**DO loop.** A range of statements invoked repetitively by a **DO** statement.

**DO variable.** A variable, specified in a **DO** statement, that is initialized or incremented prior to each occurrence of the statement or statements within a **DO** range. It is used to control the number of times the statements within the range are executed.

**DOUBLE PRECISION constant.** A constant of type real with twice the precision of the default real precision.

**dummy argument.** An entity whose name appears in the parenthesized list following the procedure name in a **FUNCTION**, **SUBROUTINE**, **ENTRY**, or statement function statement.

**dynamic dimensioning.** The process of re-evaluating the bounds of an array each time the array is referenced.

**dynamic extent.** For a directive, the lexical extent of the directive and all subprograms called from within the lexical extent.

## E

**edit descriptor.** An abbreviated keyword that controls the formatting of integer, real, or complex data.

**elemental.** Pertaining to an intrinsic operation, procedure or assignment that is applied independently to elements of an array or corresponding elements of a set of conformable arrays and scalars.

**embedded blank.** A blank that is surrounded by any other characters.

**entity.** A general term for any of the following: a program unit, procedure, operator, interface block, common block, external unit, statement function, type, named variable, expression, component of a structure, named constant, statement label, construct, or namelist group.

**environment variable.** A variable that describes the operating environment of the process.

**executable program.** A program that can be executed as a self-contained procedure. It consists of a main program and, optionally, modules, subprograms and non-Fortran external procedures.

**executable statement.** A statement that causes an action to be taken by the program; for example, to perform a calculation, test conditions, or alter normal sequential execution.

**explicit initialization.** The initialization of an object with a value stated in a data statement initial value list, block data program unit, type declaration statement, or array constructor.

**explicit interface.** For a procedure referenced in a scoping unit, the property of being an internal procedure, module procedure, intrinsic procedure, external procedure that has an interface block, recursive procedure reference in its own scoping unit, or dummy procedure that has an interface block.

**expression.** A sequence of operands, operators, and parentheses. It may be a variable, a constant, or a function reference, or it may represent a computation.

**extended-precision constant.** A processor approximation to the value of a real number that occupies 16 consecutive bytes of storage.

**external file.** A sequence of records on an input/output device. See also *internal file*.

**external name.** The name of a common block, subroutine, or other global procedure, which the linker uses to resolve references from one compilation unit to another.

**external procedure.** A procedure that is defined by an external subprogram or by a means other than Fortran.

## F

**field.** An area in a record used to contain a particular category of data.

**file.** A sequence of records. See also *external file*, *internal file*.

**file index.** See *i-node*.

**floating-point number.** A real number represented by a pair of distinct numerals. The real number is the product of the fractional part, one of the numerals, and a value obtained by raising the implicit floating-point base to a power indicated by the second numeral.

**format.** (1) A defined arrangement of such things as characters, fields, and lines, usually used for displays, printouts, or files. (2) To arrange such things as characters, fields, and lines.

**formatted data.** Data that is transferred between main storage and an input/output device according to a specified format. See also *list-directed* and *unformatted record*.

**function.** A procedure that returns the value of a single variable or an object and that usually has a single exit. See also *intrinsic procedure*, *subprogram*.

## G

**generic identifier.** A lexical token that appears in an **INTERFACE** statement and is associated with all the procedures in an interface block.

## H

**hard limit.** A system resource limit that can only be raised or lowered by using root authority, or cannot be altered because it is inherent in the system or operating environments's implementation. See also *soft limit*.

**hexadecimal.** Pertaining to a system of numbers to the base sixteen; hexadecimal digits range from 0 (zero) through 9 (nine) and A (ten) through F (fifteen).

**hexadecimal constant.** A constant, usually starting with special characters, that contains only hexadecimal digits.

**high order transformations.** A type of optimization that restructures loops.

**Hollerith constant.** A string of any characters capable of representation by XL Fortran and preceded with *nH*, where *n* is the number of characters in the string.

**host.** A main program or subprogram that contains an internal procedure is called the host of the internal procedure. A module that contains a module procedure is called the host of the module procedure.

**host association.** The process by which an internal subprogram, module subprogram, or derived-type definition accesses the entities of its host.

## I

**IPA.** Interprocedural analysis, a type of optimization that allows optimizations to be performed across procedure boundaries and across calls to procedures in separate source files.

**implicit interface.** A procedure referenced in a scoping unit other than its own is said to have an implicit interface if the procedure is an external procedure that does not have an interface block, a dummy procedure that does not have an interface block, or a statement function.

**implied DO.** An indexing specification (similar to a **DO** statement, but without specifying the word **DO**) with a list of data elements, rather than a set of statements, as its range.

**infinity.** An IEEE number (positive or negative) created by overflow or division by zero. Infinity is represented by an exponent where all the bits are 1's, and a zero fraction.

**i-node.** The internal structure that describes the individual files in the operating system. There is one i-node for each file. An i-node contains the node, type, owner, and location of a file. A table of i-nodes is stored near the beginning of a file system. Also known as *file index*.

**input/output (I/O).** Pertaining to either input or output, or both.

**input/output list.** A list of variables in an input or output statement specifying the data to be read or written. An output list can also contain a constant, an expression involving operators or function references, or an expression enclosed in parentheses.

**integer constant.** An optionally signed digit string that contains no decimal point.

**interface block.** A sequence of statements from an **INTERFACE** statement to the corresponding **END INTERFACE** statement.

**interface body.** A sequence of statements in an interface block from a **FUNCTION** or **SUBROUTINE** statement to the corresponding **END** statement.

**interference.** A situation in which two iterations within a **DO** loop have dependencies upon one another.

**internal file.** A sequence of records in internal storage. See also *external file*.

**interprocedural analysis.** See *IPA*.

**intrinsic.** Pertaining to types, operations, assignment statements, and procedures that are defined by Fortran language standards and can be used in any scoping unit without further definition or specification.

**intrinsic module.** A module that is provided by the compiler and is available to any program.

**intrinsic procedure.** A procedure that is provided by the compiler and is available to any program.

## K

**keyword.** (1) A statement keyword is a word that is part of the syntax of a statement (or directive) and that may be used to identify the statement. (2) An argument keyword specifies a name for a dummy argument.

**kind type parameter.** A parameter whose values label the available kinds of an intrinsic type.

## L

**lexical extent.** All of the code that appears directly within a directive construct.

**lexical token.** A sequence of characters with an indivisible interpretation.

**link-edit.** To create a loadable computer program by means of a linker.

**linker.** A program that resolves cross-references between separately compiled or assembled object modules and then assigns final addresses to create a single relocatable load module. If a single object module is linked, the linker simply makes it relocatable.

**list-directed.** A predefined input/output format that depends on the type, type parameters, and values of the entities in the data list.

**literal.** A symbol or a quantity in a source program that is itself data, rather than a reference to data.

**literal constant.** A lexical token that directly represents a scalar value of intrinsic type.

**load balancing.** An optimization strategy that aims at evenly distributing the work load among processors.

**logical constant.** A constant with a value of either true or false (or T or F).

**logical operator.** A symbol that represents an operation on logical expressions:

|        |                                 |
|--------|---------------------------------|
| .NOT.  | (logical negation)              |
| .AND.  | (logical conjunction)           |
| .OR.   | (logical union)                 |
| .EQV.  | (logical equivalence)           |
| .NEQV. | (logical nonequivalence)        |
| .XOR.  | (logical exclusive disjunction) |

**loop.** A statement block that executes repeatedly.

## M

**\_main.** The default name given to a main program by the compiler if the main program was not named by the programmer.

**main program.** The first program unit to receive control when a program is run. See also *subprogram*.

**master thread.** The head process of a group of threads.

**module.** A program unit that contains or accesses definitions to be accessed by other program units.

**mutex.** A primitive object that provides mutual exclusion between threads. A mutex is used cooperatively between threads to ensure that only one of the cooperating threads is allowed to access shared data or run certain application code at a time.

## N

**NaN (not-a-number).** A symbolic entity encoded in floating-point format that does not correspond to a number. See also *quiet NaN, signalling NaN*.

**name.** A lexical token consisting of a letter followed by up to 249 alphanumeric characters (letters, digits, and underscores). Note that in FORTRAN 77, this was called a symbolic name.

**named common.** A separate, named common block consisting of variables.

**namelist group name.** The first parameter in the **NAMELIST** statement that names a list of names to be used in **READ**, **WRITE**, and **PRINT** statements.

**negative zero.** An IEEE representation where the exponent and fraction are both zero, but the sign bit is 1. Negative zero is treated as equal to positive zero.

**nest.** To incorporate a structure or structures of some kind into a structure of the same kind. For example, to nest one loop (the nested loop) within another loop (the nesting loop); to nest one subroutine (the nested subroutine) within another subroutine (the nesting subroutine).

**nonexecutable statement.** A statement that describes the characteristics of a program unit, data, editing information, or statement functions, but does not cause any action to be taken by the program.

**nonexisting file.** A file that does not physically exist on any accessible storage medium.

**normal.** A floating point number that is not denormal, infinity, or NaN.

**not-a-number.** See *NaN*.

**numeric constant.** A constant that expresses an integer, real, complex, or byte number.

## O

**octal.** Pertaining to a system of numbers to the base eight; the octal digits range from 0 (zero) through 7 (seven).

**octal constant.** A constant that is made of octal digits.

**one-trip DO-loop.** A DO loop that is executed at least once, if reached, even if the iteration count is equal to 0. (This type of loop is from FORTRAN 66.)

**online processor.** In a multiprocessor machine, a processor that has been activated (brought online). The number of online processors is less than or equal to the number of physical processors actually installed in the machine. Also known as *active processor*.

**operator.** A specification of a particular computation involving one or two operands.

## P

**pad.** To fill unused positions in a field or character string with dummy data, usually zeros or blanks.

**paging space.** Disk storage for information that is resident in virtual memory but is not currently being accessed.

**PDF.** See *profile-directed feedback*.

**pointee array.** An explicit-shape or assumed-size array that is declared in an integer **POINTER** statement or other specification statement.

**pointer.** A variable that has the **POINTER** attribute. A pointer must not be referenced or defined unless it is pointer associated with a target. If it is an array, it does not have a shape unless it is pointer-associated.

**preconnected file.** A file that is connected to a unit at the beginning of execution of the executable program. Standard error, standard input, and standard output are preconnected files (units 0, 5 and 6, respectively).

**predefined convention.** The implied type and length specification of a data object, based on the initial character of its name when no explicit specification is given. The initial characters I through N imply type integer of length 4; the initial characters A through H, O through Z, \$, and \_ imply type real of length 4.

**present.** A dummy argument is present in an instance of a subprogram if it is associated with an actual argument and the actual argument is a dummy argument that is present in the invoking procedure or is not a dummy argument of the invoking procedure.

**primary.** The simplest form of an expression: an object, array constructor, structure constructor, function reference, or expression enclosed in parentheses.

**procedure.** A computation that may be invoked during program execution. It may be a function or a subroutine. It may be an intrinsic procedure, an external procedure, a module procedure, an internal procedure, a dummy procedure, or a statement function. A subprogram may define more than one procedure if it contains **ENTRY** statements.

**profile-directed feedback (PDF).** A type of optimization that uses information collected during application execution to improve performance of conditional branches and in frequently executed sections of code.

**program unit.** A main program or subprogram.

**pure.** An attribute of a procedure that indicates there are no side effects.

## Q

**quiet NaN.** A NaN (not-a-number) value that does not signal an exception. The intent of a quiet NaN is to propagate a NaN result through subsequent computations. See also *NaN*, *signalling NaN*.

## R

**random access.** An access method in which records can be read from, written to, or removed from a file in any order. See also *sequential access*.

**rank.** The number of dimensions of an array.

**real constant.** A string of decimal digits that expresses a real number. A real constant must contain a decimal point, a decimal exponent, or both.

**record.** A sequence of values that is treated as a whole within a file.

**relational expression.** An expression that consists of an arithmetic or character expression, followed by a relational operator, followed by another arithmetic or character expression.

**relational operator.** The words or symbols used to express a relational condition or a relational expression:

|      |                          |
|------|--------------------------|
| .GT. | greater than             |
| .GE. | greater than or equal to |
| .LT. | less than                |
| .LE. | less than or equal to    |
| .EQ. | equal to                 |
| .NE. | not equal to             |

**result variable.** The variable that returns the value of a function.

**return specifier.** An argument specified for a statement, such as CALL, that indicates to which statement label control should return, depending on the action specified by the subroutine in the RETURN statement.

## S

**scalar.** (1) A single datum that is not an array. (2) Not having the property of being an array.

**scale factor.** A number indicating the location of the decimal point in a real number (and, on input, if there is no exponent, the magnitude of the number).

**scope.** That part of an executable program within which a lexical token has a single interpretation.

**scope attribute.** That part of an executable program within which a lexical token has a single interpretation of a particular named property or entity.

**scoping unit.** (1) A derived-type definition. (2) An interface body, excluding any derived-type definitions and interface bodies contained within it. (3) A program unit or subprogram, excluding derived-type definitions, interface bodies, and subprograms contained within it.

**selector.** The object that is associated with the associate name in an ASSOCIATE construct.

**semantics.** The relationships of characters or groups of characters to their meanings, independent of the manner of their interpretation and use. See also *syntax*.

**sequential access.** An access method in which records are read from, written to, or removed from a file based on the logical order of the records in the file. See also *random access*.

**signalling NaN.** A NaN (not-a-number) value that signals an invalid operation exception whenever it appears as an operand. The intent of the signaling NaN is to catch program errors, such as using an uninitialized variable. See also *NaN*, *quiet NaN*.

**sleep.** The state in which a thread completely suspends execution until another thread signals it that there is work to do.

**SMP.** See *symmetric multiprocessing*.

**soft limit.** A system resource limit that is currently in effect for a process. The value of a soft limit can be raised or lowered by a process, without requiring root authority. The soft limit for a resource cannot be raised above the setting of the hard limit. See also *hard limit*.

**spill space.** The stack space reserved in each subprogram in case there are too many variables to hold in registers and the program needs temporary storage for register contents.

**specification statement.** A statement that provides information about the data used in the source program. The statement could also supply information to allocate data storage.

**stanza.** A group of lines in a file that together have a common function or define a part of the system. Stanzas are usually separated by blank lines or colons, and each stanza has a name.

**statement.** A language construct that represents a step in a sequence of actions or a set of declarations. Statements fall into two broad classes: executable and nonexecutable.

**statement function.** A name, followed by a list of dummy arguments, that is equated with an intrinsic or derived-type expression, and that can be used as a substitute for the expression throughout the program.

**statement label.** A number from one through five digits that is used to identify a statement. Statement labels can be used to transfer control, to define the range of a DO, or to refer to a FORMAT statement.

**storage association.** The relationship between two storage sequences if a storage unit of one is the same as a storage unit of the other.

**structure.** A scalar data object of derived type.

**structure component.** The part of a data object of derived-type corresponding to a component of its type.

**subobject.** A portion of a named data object that may be referenced or defined independently of other portions. It can be an array element, array section, structure component, or substring.

**subprogram.** A function subprogram or a subroutine subprogram. Note that in FORTRAN 77, a block data program unit was called a subprogram. See also *main program*.

**subroutine.** A procedure that is invoked by a **CALL** statement or defined assignment statement.

**subscript.** A subscript quantity or set of subscript quantities enclosed in parentheses and used with an array name to identify a particular array element.

**substring.** A contiguous portion of a scalar character string. (Although an array section can specify a substring selector, the result is not a substring.)

**symmetric multiprocessing (SMP).** A system in which functionally-identical multiple processors are used in parallel, providing simple and efficient load-balancing.

**synchronous.** Pertaining to an operation that occurs regularly or predictably with regard to the occurrence of a specified event in another process.

**syntax.** The rules for the construction of a statement. See also *semantics*.

## T

**target.** A named data object specified to have the **TARGET** attribute, a data object created by an **ALLOCATE** statement for a pointer, or a subobject of such an object.

**thread.** A stream of computer instructions that is in control of a process. A multithread process begins with one stream of instructions (one thread) and may later create other instruction streams to perform tasks.

**thread visible variable.** A variable that can be accessed by more than one thread.

**time slice.** An interval of time on the processing unit allocated for use in performing a task. After the interval has expired, processing unit time is allocated to another task, so a task cannot monopolize processing unit time beyond a fixed limit.

**token.** In a programming language, a character string, in a particular format, that has some defined significance.

**trigger constant.** A sequences of characters that identifies comment lines as compiler comment directives.

**type declaration statement.** A statement that specifies the type, length, and attributes of an object or function. Objects can be assigned initial values.

## U

**unformatted record.** A record that is transmitted unchanged between internal and external storage.

**Unicode.** A universal character encoding standard that supports the interchange, processing, and display of text that is written in any of the languages of the modern world. It also supports many classical and historical texts in a number of languages. The Unicode standard has a 16-bit international character set defined by ISO 10646. See also *ASCII*.

**unit.** A means of referring to a file to use in input/output statements. A unit can be connected or not connected to a file. If connected, it refers to a file. The connection is symmetric: that is, if a unit is connected to a file, the file is connected to the unit.

**unsafe option.** Any option that could result in grossly incorrect results if used in the incorrect context. Other options may result in very small variations from the default result, which is usually acceptable. Typically, using an unsafe option is an assertion that your code is not subject to the conditions that make the option unsafe.

**use association.** The association of names in different scoping units specified by a **USE** statement.

## V

**variable.** A data object whose value can be defined and redefined during the execution of an executable program. It may be a named data object, array element, array section, structure component, or substring. Note that in FORTRAN 77, a variable was always scalar and named.

## X

**XPG4.** X/Open Common Applications Environment (CAE) Portability Guide Issue 4; a document which defines the interfaces of the X/Open Common Applications Environment that is a superset of POSIX.1-1990, POSIX.2-1992, and POSIX.2a-1992 containing extensions to POSIX standards from XPG3.

## Z

**zero-length character.** A character object that has a length of 0 and is always defined.

**zero-sized array.** An array that has a lower bound that is greater than its corresponding upper bound. The array is always defined.

# INDEX

## Special characters

`_OPENMP` C preprocessor macro 30, 200

`#` compiler option 74

`-l` compiler option 75

`-B` compiler option 76

`-c` compiler option 78

`-C` compiler option 77

`-d` compiler option 80

`-D` compiler option 79

`-F` compiler option 81

`-g` compiler option 82, 331

`-I` compiler option 83

`-k` compiler option 84

`-l` compiler option 86

`-L` compiler option 85

`-N` compiler option 87

`-o` compiler option 90

`-O` compiler option 88, 267

`-O2` compiler option 88

`-O3` compiler option 88

`-O4` compiler option 88

`-O5` compiler option 89

`-p` compiler option 91

`-Q, -Q!, -Q+, -Q-` compiler options 92

`-q32` compiler option 244

`-q64` compiler option 245

`-qalias` compiler option 95, 272

`-qalign` compiler option 98

`-qarch` compiler option 29, 100, 270

`-qassert` compiler option 104, 273

`-qattr` compiler option 105, 336

`-qautodbl` compiler option 106, 353

`-qbigdata` compiler option 108

`-qcache` compiler option 29, 109, 270

`-qcclines` compiler option 111

`-qcheck` compiler option 77, 112

`-qci` compiler option 113

`-qcompact` compiler option 114

`-qcr` compiler option 115

`-qctyplss` compiler option 116

`-qdbg` compiler option 82, 118

`-qddim` compiler option 119

`-qdirective` compiler option 120

`-qdlines` compiler option 79, 123

`-qdpccomp` compiler option 124

`-qenablevmx` compiler option 125

`-qescape` compiler option 126

`-qessl` compiler option 127

`-qextern` compiler option 128

`-qextname` compiler option 129

`-qfixed` compiler option 131

`-qflag` compiler option 132

`-qfloat` compiler option 133, 257

`fltint` suboption 257

`hssngl` suboption 257

`nans` suboption 263

`nomaf` suboption 257

`rsqrt` suboption 257

`-qflttrap` compiler option 135, 258

`-qfree` compiler option 137

`-qfullpath` compiler option 138

`-qhalt` compiler option 139

`-qhot` compiler option 140, 272, 273

`-qieee` compiler option 142, 242

`-qinit` compiler option 143

`-qinitauto` compiler option 144

`-qintlog` compiler option 146

`-qintsize` compiler option 147

`-qipa` compiler option 149, 279

`-qkeepparm` compiler option 155

`-qlanglvl` compiler option 156

`-qlibansi` linker option 152

`-qlibposix` linker option 152

`-qlist` compiler option 160, 337

`-qlistopt` compiler option 161, 333

`-qlog4` compiler option 162

`-qmaxmem` compiler option 163

`-qmbcs` compiler option 165

`-qminimaltoc` compiler option 166

`-qmixed` compiler option 167

`-qmoddir` compiler option 168

`-qmodule` compiler option 169

`-qnoprint` compiler option 170

`-qnullterm` compiler option 171

`-qobject` compiler option 172

`-qonetrp` compiler option 75, 173

`-qoptimize` compiler option 88, 174

`-qpdf` compiler option 175, 276

`-qphsinfo` compiler option 179

`-qpic` compiler option 181

`-qport` compiler option 182

`-qposition` compiler option 184, 287

`-qprefetch` compiler option 185

`-qqcount` compiler option 186

`-qrealize` compiler option 187

`-qrecur` compiler option 189

`-qreport` compiler option 190, 335

`-qsaa` compiler option 192

`-qsave` compiler option 193

`-qsaveopt` compiler option 194

`-qsclock` compiler option 195

`-qshowpdf` compiler option 196

`-qsigtrap` compiler option 197, 258

`-qsmallstack` compiler option 198

`-qsmpp` compiler option 199

`-qsource` compiler option 204, 334

`-qspillsz` compiler option 87, 205

`-qstrict` compiler option 206, 267

`-qstrict_induction` compiler option 208

`-qstrictieemod` compiler option 207

`-qsuffix` compiler option 209

`-qsuppress` compiler option 210

`-qswapomp` compiler option 212

`-qtbtable` compiler option 214

`-qthreaded` compiler option 215

`-qtune` compiler option 29, 216, 270

`-qundef` compiler option 218, 236

`-qunroll` compiler option 219

`-qunwind` compiler option 220

`-qversion` compiler option 221

`-qwarn64` compiler option 247

`-qxflag=dvz` compiler option 223

`-qxflag=oldtab` compiler option 224

`-qxf77` compiler option 225

`-qxf90` compiler option 227

`-qxlines` compiler option 229

`-qxref` compiler option 231, 336

`-qzerosize` compiler option 232

`-S` compiler option 233

`-t` compiler option 234

`-u` compiler option 236

`-U` compiler option 235

`-v` compiler option 237

`-V` compiler option 238

`-w` compiler option 132, 241

`-W` compiler option 239

`-yn, -ym, -yp, -yz` compiler options 142, 242

`/etc/opt/ibmcomp/xf/9.1/xf.cfg`  
configuration file 14, 81

`/tmp` directory  
    See `TMPDIR` environment variable

`.a` files 24

`.cfg` files 24

`.f` and `.F` files 24

`.f90` suffix, compiling files with 15

`.lst` files 25

`.mod` file names 169

`.mod` files 24, 25, 34, 168

`.o` files 24, 25

`.s` files 24, 25

`.so` files 24

`.XOR` operator 225

`*` length specifiers (FORTRAN 77 extension) 342

`@PROCESS` compiler directive 27

`%REF` functions 309

`%VAL` functions 309

`#if` and other `cpp` directives 31

## Numerics

1501-229, and 1517-011 error messages 329

15xx identifiers for XL Fortran messages 326

4K suboption of `-qalign` 98

64-bit data types (FORTRAN 77 extension) 342

64-bit environment 243

## A

`a.out` file 25

actual arguments  
    definition of 367

addresses of arguments, saving 225

affinity suboption of  
    `-qsmpp=schedule` 200

`ALIAS @PROCESS` directive 95

`ALIGN @PROCESS` directive 98

alignment of BIND(C) derived types 98  
 alignment of CSECTs and large arrays for data-stripped I/O 98  
 allocatable arrays, automatic deallocation with -qxlF90=autodealloc 227  
 alphabetic character, definition of 367  
 alphanumeric, definition of 367  
 ANSI  
   checking conformance to the Fortran 90 standard 8, 39, 156  
   checking conformance to the Fortran 95 standard 8, 39, 156  
 appendold and appendunknown suboptions of -qposition 184  
 archive files 24  
 argument addresses, saving 225  
 argument promotion (integer only) for intrinsic procedures 225  
 arguments  
   definition of 367  
   passing between languages 304, 305  
   passing by reference or by value 309  
   passing null-terminated strings to C functions 171  
 arraypad suboption of -qhot 274  
 arrays  
   optimizing array language 272  
   optimizing assignments 95  
   passing between languages 307  
 arrays, initialization problems 329  
 aryovrlp suboption of -qalias 95, 272  
 as and asopt attributes of configuration file 15  
 as command, passing command-line options to 28  
 ASCII  
   definition of 367  
 assembler  
   low-level linkage conventions 312  
   source (.s) files 24, 25  
 ATTR @PROCESS directive 105  
 attribute section in compiler listing 336  
 auto suboption of -qarch 100  
 auto suboption of -qipa 149  
 auto suboption of -qsmp 199  
 auto suboption of -qtune 216  
 AUTODBL @PROCESS directive 106  
 autodealloc suboption of -qxlF90 227

## B

bash shell 12  
 BIND(C) derived types, alignment 98  
 bitwise-identical floating-point results 257  
 blankpad suboption of -qxlF77 225  
 blocked special files, interaction of XL Fortran I/O with 288  
 bolt attribute of configuration file 15  
 branches, optimizing 276  
 bss storage, alignment of arrays in 98  
 buffering run-time option  
   description 35  
   using with preconnected files 35  
 buffers, flushing 290  
 BYTE data type (FORTRAN 77 extension) 342

## C

C language and interlanguage calls 301, 304  
 C preprocessor (cpp) 29  
 C++ and Fortran in same program 302  
 calling by reference or value 309  
 calling non-Fortran procedures 301  
 carriage return character 115  
 CCLINES @PROCESS 111  
 character constants and typeless constants 116  
 character data, passing between languages 306  
 character special files, interaction of XL Fortran I/O with 288  
 character-count edit descriptor (FORTRAN 77 extension) 342  
 CHECK @PROCESS directive 77, 112  
 check\_fpscr.f sample file 262  
 chunk  
   definition of 368  
 CI @PROCESS directive 113  
 cleanpdf command 176  
 cnvrr run-time option 37  
 code attribute of configuration file 15  
 code generation for different systems 29  
 code optimization 9, 265  
 com suboption of -qarch 100  
 command line, specifying options on 27  
 command-line options  
   *See* compiler options  
 COMPACT @PROCESS directive 114  
 compexgcc suboption of -qfloat 133  
 compilation order 23  
 compilation unit epilogue section in compiler listing 337  
 compiler listings 333  
   *See also* listings  
 compiler options for controlling 61  
 compiler options  
   *See also* the individual options listed under Special Characters at the start of the index  
   deprecated 71  
   descriptions 73  
   for compatibility 62  
   for controlling input to the compiler 52  
   for controlling listings and messages 61  
   for controlling the compiler internal operation 71  
   for debugging and error checking 58, 59  
   for floating-point processing 70  
   for linking 70  
   for performance optimization 54  
   obsolete or not recommended 71  
   scope and precedence 26  
   section in compiler listing 333  
   specifying in the source file 27  
   specifying on the command line 27  
   specifying the locations of output files 54  
   summary 51  
 compiler options for 64-bit 243

compiling  
   cancelling a compilation 24  
   description of how to compile a program 21  
   problems 328  
   SMP programs 23  
 conditional branching optimization 276  
 conditional compilation 29  
 conditional vector merge intrinsic functions (FORTRAN 77 extension) 342  
 configuration file 14, 24, 81  
 conflicting options  
   -C interferes with -qhot 77  
   -qautodbl overrides -qrealize 107  
   -qdpc is overridden by -qautodbl and -qrealize 187  
   -qflag overrides -qlanglvl and -qsaa 132  
   -qhalt is overridden by -qnoobject 172  
   -qhalt overrides -qobject 172  
   -qhot is overridden by -C 140  
   -qintsize overrides -qlog4 162  
   -qlanglvl is overridden by -qflag 157  
   -qlog4 is overridden by -qintsize 162  
   -qnoobject overrides -qhalt 139  
   -qobject is overridden by -qhalt 139  
   -qrealize is overridden by -qautodbl 107, 188  
   -qrealize overrides -qdpc 187  
   -qsaa is overridden by -qflag 192  
   @PROCESS overrides command-line setting 26  
   command-line overrides configuration file setting 26  
   specified more than once, last one takes effect 26  
 conformance checking 8, 156, 192  
 control and status register for floating point 261  
 conversion errors 37  
 core file 258, 331  
 cost model for loop transformations 273  
 could not load program (error message) 328  
 cpp command 29  
 cpp, cppoptions, and cppsuffix attributes of configuration file 15  
 cpu\_time\_type run-time option 37  
 CRAY functions (FORTRAN 77 extension)  
   conditional vector merge intrinsics 342  
   date and time service and utility functions 342  
 CRAY pointer (FORTRAN 77 extension), XL Fortran equivalent 342  
 cross-reference section in compiler listing 336  
 crt attribute of configuration file 15  
 crt\_64 attribute of configuration file 15  
 CSECTs, alignment of 98  
 csh shell 12  
 CTYPLSS @PROCESS directive 116  
 customizing configuration file (including default compiler options) 14

CVMGx intrinsic functions (FORTRAN 77 extension) 342

## D

data limit 328  
data striping  
  -align required for improved performance 98  
data types in Fortran, C 305  
date and time functions (FORTRAN 77 extension) 342  
DBG @PROCESS directive 82, 118  
dbl, dbl4, dbl8, dblpad, dblpad4, dblpad8 suboptions of -qautodbl 106  
DDIM @PROCESS directive 119  
debugger support 9  
debugging 325  
  compiler options for 58  
  using path names of original files 138  
default\_recl run-time option 38  
defaultmsg attribute of configuration file 15  
defaults  
  customizing compiler defaults 14  
  search paths for include and .mod files 83  
  search paths for libraries 13  
delays run-time option 45  
deprecated compiler options 71  
deps suboption of -qassert 104  
DIRECTIVE @PROCESS directive 120  
directives  
  NEW 341  
disassembly listing  
  from the -S compiler option 233  
disk space, running out of 329  
disk striping  
  See data striping  
DLINES @PROCESS directive 79, 123  
documentation, online formats 9  
double-precision values 250, 252  
DPC @PROCESS directive 124  
dummy argument  
  definition of 369  
dynamic dimensioning of arrays 119  
dynamic extent, definition of 369  
dynamic linking 32  
dynamic suboption of  
  -qsmpt=schedule 201

## E

E error severity 325  
edit descriptors (B, O, Z), differences between F77 and F90 225  
edit descriptors (G), difference between F77 and F90 225  
editing source files 21  
emacs text editor 21  
enable suboption of -qfltrap 135, 260  
end-of-file, writing past 225  
ENTRY statements, compatibility with previous compiler versions 225  
environment problems 328

environment variables

  compile time 12  
    LANG 12  
    NLSPATH 12  
    PDFDIR 14  
    TMPDIR 14  
  LD\_LIBRARY\_PATH 13  
  LD\_RUN\_PATH 13  
  OpenMP  
    OMP\_DYNAMIC 47  
    OMP\_NESTED 48  
    OMP\_NUM\_THREADS 48  
    OMP\_SCHEDULE 49  
  run-time  
    LD\_LIBRARY\_PATH 49  
    LD\_RUN\_PATH 49  
    PDFDIR 14  
    TMPDIR 49  
    XLFRTOPTS 35  
    XLSMPOPTS 43  
    XLFSCRATCH\_unit 14  
    XLFUNIT\_unit 14  
  eof, writing past 225  
  epilogue sections in compiler listing 337  
  err\_recovery run-time option 38  
  error checking, compiler options for 58  
  error messages 325  
    1501-229 329  
    1517-011 329  
    compiler options for controlling 61  
    explanation of format 326  
    in compiler listing 334  
  erroreof run-time option 38  
  ESCAPE @PROCESS directive 126  
  example programs  
    See sample programs  
  exception handling 49, 251  
    for floating point 135, 257  
    installing an exception handler 258  
  exclusive or operator 225  
  executable files 25  
  executing a program 33  
  executing the compiler 21  
  exits suboption of -qipa 149  
  explicit interfaces 310  
  extended-precision values 253  
  extensions to FORTRAN 77, list of common ones 342  
  external names  
    in the run-time environment 352  
  EXTNAME @PROCESS directive 129

## F

f77 command  
  description 21  
  level of Fortran standard compliance 22  
f90 suffix 15  
fexcp.h include file 259  
fhandler.F sample file 262  
file positioning 287  
file table section in compiler listing 337  
files  
  editing source 21  
  I/O formats 285  
  input 24

files (*continued*)

  names 286  
  output 25  
  permissions 289  
  using suffixes other than .f for source files 15  
FIPS FORTRAN standard, checking conformance to 8  
FIXED @PROCESS directive 131  
FLAG @PROCESS directive 132  
FLOAT @PROCESS directive 133  
floating-point  
  exception handling 49  
  exceptions 135, 257  
  processing 249  
  optimizing 257, 271  
floating-point status and control register 261  
fltint suboption of -qfloat 133  
FLTRAP @PROCESS directive 135, 258  
fltrap\_handler.c and fltrap\_test.f sample files 262  
flushing I/O buffers 290  
fold suboption of -qfloat 133  
formats, file 285  
fort.\* default file names 286, 290  
fort77 command  
  description 21  
Fortran 2003 features 39  
Fortran 2003 iostat\_end behaviour 38  
FORTRAN 77 extensions, list of common ones 342  
Fortran 90  
  compiling programs written for 22  
  fpdt.h and fpdc.h include files 254  
  fpgets and fpsets service and utility subroutines 261  
  fppv and fppk attributes of configuration file 15  
  fpscr register 261  
  fpstat array 261  
  FREE @PROCESS directive 137  
  fsuffix attribute of configuration file 15  
  full suboption of -qbttable 214  
FULLPATH @PROCESS directive 138  
functions  
  linkage convention for calls 320  
  return values 311

## G

G edit descriptor, difference between F77 and F90 225  
gcr attribute of configuration file 15  
gcr\_64 attribute of configuration file 15  
gedit77 suboption of -qxf77 225  
generating code for different systems 29  
get\_round\_mode procedure 254  
GETENV intrinsic procedure 286  
guided suboption of  
  -qsmpt=schedule 201

## H

HALT @PROCESS directive 139  
hardware, compiling for different types of 29  
header section in compiler listing 333  
hexint and nohexint suboptions of -qport 182  
hot attribute of configuration file 15  
hotlist suboption of -qreport 190  
hsflt suboption of -qfloat 133, 352  
HTML documentation 9

## I

I error severity 325  
i-node 41  
I/O  
    *See* input/output  
IBM Distributed Debugger 9  
IEEE @PROCESS directive 142, 242  
IEEE arithmetic 249  
implicitly connected files 286  
imprecise suboption of -qflttrap 135  
include directory 34  
include files fpdt.h and fpdc.h 254  
include\_32 attribute of configuration file 15  
include\_64 attribute of configuration file 15  
inexact suboption of -qflttrap 135  
infinity values 250  
informational message 325  
INIT @PROCESS directive 143  
initial file position 287  
initialize arrays, problems 329  
inline suboption of -qipa 149  
inlining 92, 277  
input files 24  
input/output 249  
    from two languages in the same program 302  
    increasing throughput with data striping 98  
    redirection 288  
    run-time behavior 35  
    when unit is positioned at end-of-file 225  
    XL Fortran implementation details 285  
installation problems 328  
installing the compiler 11  
intarg suboption of -qxlf77 225  
integer arguments of different kinds to intrinsic procedures 225  
integer POINTER (FORTRAN 77 extension) 342  
INTENT attribute 311  
interlanguage calls 301, 309  
    arrays 307  
    C++ 302  
    character types 306  
    corresponding data types 305  
    input and output 302  
    low-level linkage conventions 312  
    pointers 308  
internal limits for the compiler 361

interprocedural analysis (IPA) 149  
INTLOG @PROCESS directive 146  
intptr suboption of -qalias 95  
intrinsic procedures accepting integer arguments of different kinds 225  
intrinths run-time option 39  
INTSIZE @PROCESS directive 147  
intxor suboption of -qxlf77 225  
invalid suboption of -qflttrap 135  
invoking a program 33  
invoking the compiler 21  
iostat\_end run-time option 38  
ipa attribute of configuration file 15  
irand routine, naming restriction for 33  
ISO  
    checking conformance to the Fortran 90 standard 8, 39, 156  
    checking conformance to the Fortran 95 standard 8, 39, 156  
isolated suboption of -qipa 149  
itercnt suboption of -qassert 104

## K

kind type parameters 147, 187  
ksh shell 12

## L

L error severity 325  
LANG environment variable 12  
LANGLVL @PROCESS directive 156  
langlvl run-time option 39  
language support 7  
language-level error 325  
large and small suboptions of -qpdc 181  
LC\_\* national language categories 13  
ld and ldopt attributes of configuration file 15  
ld command  
    passing command-line options to 28  
LD\_LIBRARY\_PATH environment variable 13, 49  
LD\_RUN\_PATH environment variable 13, 49  
leadzero suboption of -qxlf77 225  
level of XL Fortran, determining 20  
level suboption of -qipa 149  
lexical extent, definition of 371  
lib\*.so library files 24, 86  
libraries 24  
    default search paths 13  
    shared 351  
libraries attribute of configuration file 15  
library path environment variable 328  
libxlf90\_r.dylib 22  
libxlf90\_r.so 22  
libxlf90\_r.so library 20, 34  
libxlf90\_t.so 22  
libxlf90\_t.so library 20  
libxlsmp.so library 34  
limit command 328  
limit suboption of -qipa 149, 279  
limits internal to the compiler 361  
line feed character 115

linker options 70  
    -qlibansi 152  
    -qlibposix 152  
linking 32  
    dynamic 32  
    problems 329  
    static 32  
links, interaction of XL Fortran I/O with 288  
LIST @PROCESS directive 160  
list suboption of -qipa 149, 150  
listing files 25  
listing options 61  
LISTOPT @PROCESS directive 161  
locale, setting at run time 34  
LOG4 @PROCESS directive 162  
long variable names (FORTRAN 77 extension) 342  
loops, optimizing 272  
lower case (FORTRAN 77 extension) 342  
lowfreq suboption of -qipa 149

## M

m suboption of -y 242  
machines, compiling for different types 29, 100  
macro expansion 29  
macro, \_OPENMP C preprocessor 30, 200  
maf suboption of -qfloat 133, 206  
main, restriction on use as a Fortran name 301  
make command 74  
makefiles  
    configuration file as alternative for default options 15  
    copying modified configuration files along with 15  
malloc system routine 107  
MAXMEM @PROCESS directive 163  
MBCS @PROCESS directive 165  
mclock routine, naming restrictions for 33  
mcr\_64 attribute of configuration file 15  
memory management optimizations 272  
message suppression 210  
messages  
    1501-053 error message 329  
    1501-229 error message 329  
    1517-011 error message 329  
    catalog files for 327  
    compiler options for controlling 61  
    copying message catalogs to another system 327  
    selecting the language for run-time messages 34  
migrating 8  
    from other systems 339  
minus infinity, representation of 250  
minus suboption of -qieee 142  
missing suboption of -qipa 149  
MIXED @PROCESS directive 167, 235  
mixing integers and logicals (FORTRAN 77 extension) 342  
mod and nomod suboptions of -qport 182

mod file names, intrinsic 169  
mod files 24, 25, 168  
module procedures, external names  
  corresponding to 301  
modules, effect on compilation order 23  
mon.out file 24  
multconn run-time option 40  
multconnio run-time option 41

## N

n suboption of -y 242  
name conflicts, avoiding 33  
namelist run-time option 41  
naming conventions for external  
  names 301  
NaN values  
  and infinities 250  
  specifying with -qinitauto compiler  
  option 144  
nans suboption of -qfloat 133  
national language support  
  at run time 34  
  compile time environment 12  
nearest suboption of -qieee 142  
negative infinity, representation of 250  
nested\_par suboption of -qsmp 199  
network file system (NFS)  
  using the compiler on a 11  
Network Install Manager 11  
NEW compiler directive 341  
NFS  
  *See* network file system  
NIM (Network Install Manager) 11  
NLSPATH environment variable  
  compile time 12  
nlwidth run-time option 42  
noauto suboption of -qsmp 199  
nodblpad suboption of -qautodbl  
  *See* none suboption instead  
nodeps suboption of -qassert 104  
noinline suboption of -qipa 149  
none suboption of -qautodbl 106  
none suboption of -qhtable 214  
nonested\_par suboption of -qsmp 199  
noobject suboption of -qipa 149  
noomp suboption of -qsmp 199  
noopt suboption of -qsmp 200  
norec\_locks suboption of -qsmp 200  
null-terminated strings, passing to C  
  functions 171, 306  
NULLTERM @PROCESS directive 171

## O

OBJECT @PROCESS directive 172  
object files 24, 25  
object suboption of -qipa 149  
obsolete compiler options 71  
oldboz suboption of -qxlf77 225  
omp suboption of -qsmp 199  
OMP\_DYNAMIC environment  
  variable 47  
OMP\_NESTED environment variable 48  
OMP\_NUM\_THREADS environment  
  variable 48

OMP\_SCHEDULE environment  
  variable 49  
ONETRIP @PROCESS directive 75, 173  
online compiler help 9  
online documentation 9  
OpenMP environment variables 47  
opt suboption of -qsmp 200  
optimization 9, 265  
  compiler options for 54  
  for floating-point arithmetic 257  
  levels 267  
OPTIMIZE @PROCESS directive 88, 174  
OPTIONAL attribute 311  
options attribute of configuration file 15  
options section in compiler listing 333  
osuffix attribute of configuration file 15  
output files 25  
overflow suboption of -qfltrap 135

## P

p suboption of -y 242  
pad setting, changing for internal,  
  direct-access and stream-access  
  files 225  
padding of data types with -qautodbl  
  option 353  
paging space  
  running out of 329  
parallel execution options 44  
parameters  
  *See* arguments  
parthds run-time option 44  
parthresold run-time option 46  
partition suboption of -qipa 149  
Pascal language and interlanguage  
  calls 301  
path name of source files, preserving  
  with -qfullpath 138  
PDF documentation 9  
PDFDIR environment variable 14  
pdfname suboption of -qipa 149, 151  
performance of floating-point  
  arithmetic 257  
performance of real operations, speeding  
  up 107, 187  
performance tuning options 45  
permissions of files 289  
persistent suboption of -qxlf77 225  
PHSINFO @PROCESS directive 179  
pipes, interaction of XL Fortran I/O  
  with 288  
platform, compiling for a specific  
  type 100  
plus infinity, representation of 250  
plus suboption of -qieee 142  
pointers (Fortran 90) and -qinit compiler  
  option 143  
pointers (integer POINTER) (FORTRAN  
  77 extension) 342  
PORT @PROCESS directive 182  
portability 339  
porting to XL Fortran 339  
POSITION @PROCESS directive 184,  
  287  
position of a file after an OPEN  
  statement 287

positive infinity, representation of 250  
POSIX pthreads  
  API support 23  
  run-time libraries 34  
postmortem.f sample file 262  
POWER3, POWER4, POWER5, or  
  PowerPC systems 100  
  compiling programs for 29  
ppc suboption of -qarch 100  
ppc64 suboption of -qarch 101  
ppc64gr suboption of -qarch 101  
ppc64grsq suboption of -qarch 101  
ppc970 suboption of -qtune 216  
ppcgr suboption of -qarch 100  
precision of real data types 107, 187  
preconnected files 286  
preprocessing Fortran source with the C  
  preprocessor 29  
problem determination 325  
procedure calls to other languages  
  *See* subprograms in other languages,  
  calling  
prof command 25  
profilefreq run-time option 47  
profiling data files 25  
Program Editor 21  
promoting integer arguments to intrinsic  
  procedures 225  
promotion of data types with -qautodbl  
  option 353  
pseudo-devices, interaction of XL Fortran  
  I/O with 288  
pteovrlp suboption of -qalias 95  
pthread library module 349  
pure suboption of -qipa 149  
pwr3 suboption of -qarch 101  
pwr3 suboption of -qtune 216  
pwr4 suboption of -qarch 101  
pwr5 suboption of -qarch 101

## Q

Q (character-count) edit descriptor  
  (FORTRAN 77 extension) 342  
QCOUNT @PROCESS directive 186  
qdirectstorage compiler option 122  
quiet NaN 144, 250  
quiet NaN suboption of -qfltrap 135

## R

rand routine, naming restriction for 33  
random run-time option 42  
READ statements past end-of-file 225  
README.xlf file 11  
real arithmetic 249  
REAL data types 107  
REAL(16) values 253  
REAL(4) and REAL(8) values 250, 252  
REALSIZE @PROCESS directive 187  
rec\_locks suboption of -qsmp 200  
record lengths 289  
RECUR @PROCESS directive 189  
recursion 189, 193  
redirecting input/output 288  
reference, passing arguments by 309

- register flushing 155
- related documentation 4
- REPORT @PROCESS directive 190
- resetpdf command 176
- return code
  - from compiler 326
  - from Fortran programs 326
- rounding 254
  - rounding errors 256
  - rounding mode 254, 256
- rpm command 20
- rrm suboption of -qfloat 133, 206
- rsqrt suboption of -qfloat 133
- run time
  - exceptions 49
  - options 35
- run-time
  - libraries 24
  - problems 330
- run-time environment
  - external names in 352
- running a program 33
- running the compiler 21
- runtime suboption of
  - qsmp=schedule 201

**S**

- S error severity 325
- SAA @PROCESS directive 192
- SAA FORTRAN definition, checking
  - conformance to 8
- safe suboption of -qipa 149
- sample programs 345
  - calling C functions from Fortran 306
  - floating-point exception
    - handling 262
  - for SMP 346
  - notes on using 4
- SAVE @PROCESS directive 193
- schedule run-time option 43
- schedul suboption of -qsmp 200
- scratch file directory
  - See TMPDIR environment variable
- scratch\_vars run-time option 14, 42, 291
- segmentation fault 175
- seqthreshold run-time option 46
- setlocale libc routine 34
- setrteopts service and utility
  - procedure 35
- severe error 325
- sh shell 12
- shared libraries 351
- shared object files 24
- side-effects, definition of 149
- SIGFPE signal 258
- SIGN intrinsic, effect of
  - qxl90=signedzero on 227
- signal handling 49
  - for floating point 257
  - installing an exception handler 258
- signaling NaN 250, 263
- signedzero suboption of -qxl90 227
- SIGTRAP signal 49, 258
- single-precision values 250, 252
- small and large suboptions of -qplic 181
- small suboption of -qbttable 214

- SMP
  - programs, compiling 23
  - sample programs 346
- smplibraries attribute of configuration file 15
- smplist suboption of -qreport 190
- softeof suboption of -qxl77 225
- SOURCE @PROCESS directive 204
- source file options 27
- source files 24
  - allowing suffixes other than .f 15
  - preserving path names for
    - debugging 138
  - specifying options in 27
- source section in compiler listing 334
- source-code conformance checking 8
- source-level debugging support 9
- space problems 328
- special files, interaction of XL Fortran I/O
  - with 288
- SPILLSIZE @PROCESS directive 87, 205
- spins run-time option 45
- ssuffix attribute of configuration file 15
- stack 313
  - limit 328
- stack run-time option 45
- standard error, input, and output
  - streams 286
- star length specifiers 342
- static linking 32
- static storage, alignment of arrays in 98
- static suboption of -qsmp=schedule 201
- status and control register for floating
  - point 261
- std suboption of -qalias 95
- stderr, stdin, and stdout streams 286
- stdexits suboption of -qipa 149
- storage limits 328
- storage relationship between data
  - objects 353
- storage-associated arrays, performance
  - implications of 95
- STRICT @PROCESS directive 206
- strictieemod @PROCESS directive 207
- strictnmaf suboption of -qfloat 133
- strings, passing to C functions 171, 306
- subprogram calls to other languages
  - See subprograms in other languages,
    - calling
- subprograms in other languages,
  - calling 301, 304
- suffix, allowing other than .f on source
  - files 15
- suffixes for source files 209
- summary of compiler options 51
- Sun pointer (FORTRAN 77 extension), XL
  - Fortran equivalent 342
- SWAPOMP @PROCESS directive 212
- symbolic debugger support 9
- symbolic links, interaction of XL Fortran
  - I/O with 288
- syntax diagrams and statements 2
- system problems 328

**T**

- target machine, compiling for 100
- tctl command 288
- temporary arrays, reducing 95, 272
- temporary file directory 14
- temporary files
  - See /tmp directory
- text editors 21
- TextEdit text editor 21
- threads, controlling 39
- threshold suboption of -qipa 149
- threshold suboption of -qsmp 201
- throughput for I/O, increasing with data
  - striping 98
- time and date functions (FORTRAN 77
  - extension) 342
- times routine, naming restriction for 33
- TMPDIR environment variable 49, 329
  - compile time 14
- Trace/BPT trap 49, 258
- traceback listing 197, 259, 331
- transformation report section in compiler
  - listing 335
- trigger\_constant
  - \$OMP 199
  - IBM\* 120
  - IBMP 199
  - IBMT 215
  - setting values 120
  - SMP\$ 199
- trigraphs 31
- tuning performance
  - See optimization
- typeless constants (FORTRAN 77
  - extension) 342
- typeless constants and character
  - constants 116
- typestmt and notypestmt suboptions of
  - qport 182

**U**

- U error severity 325
- ulimit command 328
- UNDEF @PROCESS directive 218, 236
- underflow suboption of -qfltrap 135
- Unicode data 165
- unit\_vars run-time option 14, 42, 290
- UNIVERSAL setting for locale 165
- unknown suboption of -qipa 149
- unrecoverable error 325
- unrolling DO LOOPS 219
- unrolling loops 274
- UNWIND @PROCESS directive 220
- use attribute of configuration file 15
- usrthds run-time option 45
- UTF-8 encoding for Unicode data 165
- uwidth run-time option 42

**V**

- value relationships between data
  - objects 353
- value, passing arguments by 309
- vi text editor 21

## W

W error severity 325  
warning error 325  
what command 20  
WRITE statements past end-of-file 225

## X

XFLAG(OLDTAB) @PROCESS  
  directive 224  
xl\_ieee exception handler 259  
xl\_ieee.F and xl\_ieee.c sample  
  files 262  
xl\_sigdump exception handler 259  
xl\_trbk exception handler 259  
xl\_trbk library procedure 331  
xl\_trbk\_test.f sample file 262  
xl\_trce exception handler 197, 259  
xl\_trcedump exception handler 259  
xlf attribute of configuration file 15  
xlf command  
  description 21  
  level of Fortran standard  
  compliance 22  
xlf\_r command  
  description 21  
  for compiling SMP programs 23  
  level of Fortran standard  
  compliance 22  
xlf.cfg configuration file 81  
XLF77 @PROCESS directive 225  
XLF90 @PROCESS directive 227  
xlf90 command  
  description 21  
  level of Fortran standard  
  compliance 22  
xlf90\_r command  
  description 21  
  for compiling SMP programs 23  
  level of Fortran standard  
  compliance 22  
xlf95 command  
  description 21  
xlf95\_r command  
  description 21  
  for compiling SMP programs 23  
  level of Fortran standard  
  compliance 22  
xlfopt attribute of configuration file 15  
XLFRTEOPTS environment variable 35  
XLFSCRATCH\_unit environment  
  variable 14, 42, 291  
XLFUNIT\_unit environment variable 14,  
  42, 290  
XLINES @PROCESS 229  
XLSMPOPTS environment variable 43  
XOR 225  
XREF @PROCESS directive 231  
xrf\_messages run-time option 42

## Y

yields run-time option 45

## Z

z suboption of -y 242  
zero suboption of -qieec 142  
zerodivide suboption of -qflttrap 135  
zeros (leading), in output 225  
ZEROSIZE @PROCESS directive 232







Program Number: 5724-K76

SC09-7948-01

