

XL Fortran Advanced Edition for Mac OS X



# User's Guide

*Version 8.1*



XL Fortran Advanced Edition for Mac OS X



# User's Guide

*Version 8.1*

**Note!**

Before using this information and the product it supports, be sure to read the general information under "Notices" on page 293.

**First Edition (December 2003)**

This edition applies to IBM XL Fortran Advanced Edition Version 8.1 for Mac OS X (Program 5724-G13) 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.

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](mailto: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, 2003. All rights reserved.

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

---

# Contents

<b>Figures</b> . . . . .	<b>vii</b>
--------------------------	------------

---

<b>Using XL Fortran Features</b> . . . . .	<b>1</b>
--------------------------------------------	----------

<b>Introduction</b> . . . . .	<b>3</b>
-------------------------------	----------

How to Use This Book . . . . .	3
How to Read the Syntax Diagrams and Statements	4
Notes on the Examples in This Book . . . . .	6
Notes on the Path Names in This Book . . . . .	6
Notes on the Terminology in This Book . . . . .	6
Related Documentation . . . . .	6
XL Fortran and Operating System Publications . . . . .	6
Standards Documents . . . . .	6

<b>Overview of XL Fortran Features</b> . . . . .	<b>9</b>
--------------------------------------------------	----------

Hardware and Operating-System Support . . . . .	9
Language Support . . . . .	9
Migration Support . . . . .	9
Source-Code Conformance Checking . . . . .	10
Highly Configurable Compiler . . . . .	10
Diagnostic Listings . . . . .	11
Symbolic Debugger Support . . . . .	11
Program Optimization . . . . .	11
Online Documentation . . . . .	11

<b>Setting Up and Customizing XL Fortran</b> <b>13</b>
--------------------------------------------------------

Where to Find Installation Instructions . . . . .	13
Correct Settings for Environment Variables . . . . .	13
Environment Variable Basics . . . . .	13
PDFDIR: Specifying the Directory for PDF Profile Information . . . . .	14
TMPDIR: Specifying a Directory for Temporary Files . . . . .	14
XLFSRATCH_unit: Specifying Names for Scratch Files . . . . .	14
XLFUNIT_unit: Specifying Names for Implicitly Connected Files . . . . .	14
Customizing the Configuration File . . . . .	15
Attributes . . . . .	15
What a Configuration File Looks Like . . . . .	17
Using XL Fortran with Xcode and Project Builder . . . . .	19
Configure the Xcode IDE . . . . .	19
Setting Up XL Fortran to Use Xcode Projects . . . . .	19
Using XL Fortran with Project Builder . . . . .	20
Setting the CC Variable . . . . .	21

<b>Editing, Compiling, Linking, and Running XL Fortran Programs</b> . . . . .	<b>23</b>
-------------------------------------------------------------------------------	-----------

Editing XL Fortran Source Files . . . . .	23
Compiling XL Fortran Programs . . . . .	23
Compiling Fortran 90 or Fortran 95 Programs . . . . .	24
Compilation Order for Fortran Programs . . . . .	25
Canceling a Compilation . . . . .	25
XL Fortran Input Files . . . . .	25

XL Fortran Output Files . . . . .	26
Scope and Precedence of Option Settings . . . . .	27
Specifying Options on the Command Line . . . . .	28
Specifying Options in the Source File . . . . .	29
Passing Command-Line Options to the "ld" or "as" Commands . . . . .	29
Compiling for PowerPC Systems . . . . .	30
Passing Fortran Files through the C Preprocessor	31
cpp Directives for XL Fortran Programs . . . . .	31
Passing Options to the C Preprocessor . . . . .	32
Avoiding Preprocessing Problems . . . . .	32
Linking XL Fortran Programs . . . . .	32
Compiling and Linking in Separate Steps . . . . .	32
Passing Options to the ld Command . . . . .	33
Dynamic Linking . . . . .	33
Avoiding Naming Conflicts during Linking . . . . .	33
Running XL Fortran Programs . . . . .	34
Canceling Execution . . . . .	34
Compiling and Executing on Different Systems	34
Setting Run-Time Options . . . . .	34
Other Environment Variables That Affect Run-Time Behavior . . . . .	41
XL Fortran Run-Time Exceptions . . . . .	41

<b>XL Fortran Compiler-Option Reference</b> <b>43</b>
-------------------------------------------------------

Summary of the XL Fortran Compiler Options . . . . .	43
Options That Control Input to the Compiler . . . . .	44
Options That Specify the Locations of Output Files . . . . .	45
Options for Performance Optimization . . . . .	46
Options for Error Checking and Debugging . . . . .	49
Options That Control Listings and Messages . . . . .	51
Options for Compatibility . . . . .	53
Option for New Language Extensions . . . . .	58
Options for Floating-Point Processing . . . . .	59
Options That Control Linking . . . . .	59
Options That Control the Compiler Internal Operation . . . . .	60
Options That Are Obsolete or Not Recommended	61
Detailed Descriptions of the XL Fortran Compiler Options . . . . .	62
-# Option . . . . .	63
-1 Option . . . . .	64
-B Option . . . . .	65
-C Option . . . . .	66
-c Option . . . . .	67
-D Option . . . . .	68
-d Option . . . . .	69
-F Option . . . . .	70
-g Option . . . . .	71
-I Option . . . . .	72
-k Option . . . . .	73
-L Option . . . . .	74
-l Option . . . . .	75
-N Option . . . . .	76

-O Option . . . . .	77
-o Option . . . . .	79
-p Option . . . . .	80
-qalias Option . . . . .	81
-qalign Option . . . . .	84
-qarch Option . . . . .	86
-qassert Option . . . . .	88
-qattr Option . . . . .	89
-qautodbl Option . . . . .	90
-qcache Option . . . . .	92
-qcclines Option . . . . .	94
-qcharlen Option . . . . .	95
-qcheck Option . . . . .	96
-qci Option . . . . .	97
-qcommon Option . . . . .	98
-qcompact Option . . . . .	99
-qcr Option . . . . .	100
-qctypsls Option . . . . .	101
-qdbg Option . . . . .	103
-qddim Option . . . . .	104
-qdirective Option . . . . .	105
-qdlines Option . . . . .	107
-qdpc Option . . . . .	108
-qescape Option . . . . .	109
-qextern Option . . . . .	110
-qextname Option . . . . .	111
-qfixed Option . . . . .	113
-qflag Option . . . . .	114
-qfloat Option . . . . .	115
-qflttrap Option . . . . .	117
-qfree Option . . . . .	119
-qfullpath Option . . . . .	120
-qhalt Option . . . . .	121
-qhot Option . . . . .	122
-qiee Option . . . . .	123
-qinit Option . . . . .	124
-qinitauto Option . . . . .	125
-qintlog Option . . . . .	127
-qintsize Option . . . . .	128
-qipa Option . . . . .	130
-qkeepparm Option . . . . .	135
-qlanglvl Option . . . . .	136
-qlibansi Option . . . . .	138
-qlibposix Option . . . . .	139
-qlist Option . . . . .	140
-qlistopt Option . . . . .	141
-qlog4 Option . . . . .	142
-qmaxmem Option . . . . .	143
-qmbsc Option . . . . .	145
-qmixed Option . . . . .	146
-qmoddir Option . . . . .	147
-qnoprint Option . . . . .	148
-qnullterm Option . . . . .	149
-qobject Option . . . . .	150
-qonetrip Option . . . . .	151
-qoptimize Option . . . . .	152
-qpdf Option . . . . .	153
-qphsinfo Option . . . . .	156
-qpic Option . . . . .	157
-qport Option . . . . .	158
-qposition Option . . . . .	159

-qprefetch Option . . . . .	160
-qqcount Option . . . . .	161
-qrealsize Option . . . . .	162
-qrecur Option . . . . .	164
-qreport Option . . . . .	165
-qsa Option . . . . .	166
-qsave Option . . . . .	167
-qsigtrap Option . . . . .	168
-qsmallstack Option . . . . .	169
-qsource Option . . . . .	170
-qspillsize Option . . . . .	171
-qstrict Option . . . . .	172
-qstrictieemod Option . . . . .	173
-qstrict_induction Option . . . . .	174
-qsuffix Option . . . . .	175
-qsuppress Option . . . . .	176
-qthreaded Option . . . . .	178
-qtune Option . . . . .	179
-qundef Option . . . . .	180
-qunroll Option . . . . .	181
-qunwind Option . . . . .	182
-qxflag=oldtab Option . . . . .	183
-qxlf77 Option . . . . .	184
-qxlf90 Option . . . . .	186
-qxlines Option . . . . .	188
-qxref Option . . . . .	190
-qzerosize Option . . . . .	191
-t Option . . . . .	192
-U Option . . . . .	193
-u Option . . . . .	194
-v Option . . . . .	195
-V Option . . . . .	196
-W Option . . . . .	197
-w Option . . . . .	198
-y Option . . . . .	199

## XL Fortran Floating-Point Processing 201

IEEE Floating-Point Overview . . . . .	201
Compiling for Strict IEEE Conformance . . . . .	201
IEEE Single- and Double-Precision Values . . . . .	202
IEEE Extended-Precision Values . . . . .	202
Infinities and NaNs . . . . .	202
Exception-Handling Model . . . . .	203
Hardware-Specific Floating-Point Overview . . . . .	204
Single- and Double-Precision Values . . . . .	204
Extended-Precision Values . . . . .	205
How XL Fortran Rounds Floating-Point Calculations . . . . .	206
Selecting the Rounding Mode . . . . .	206
Minimizing Rounding Errors . . . . .	208
Minimizing Overall Rounding . . . . .	208
Delaying Rounding until Run Time . . . . .	208
Ensuring that the Rounding Mode is Consistent . . . . .	208
Duplicating the Floating-Point Results of Other Systems . . . . .	209
Maximizing Floating-Point Performance . . . . .	209
Detecting and Trapping Floating-Point Exceptions . . . . .	209
Compiler Features for Trapping Floating-Point Exceptions . . . . .	210
Installing an Exception Handler . . . . .	210

Controlling the Floating-Point Status and Control Register . . . . .	212
xlf_fp_util Procedures . . . . .	212
fpgets and fpsets Subroutines . . . . .	213
Sample Programs for Exception Handling . . . . .	214
Causing Exceptions for Particular Variables . . . . .	215
Minimizing the Performance Impact of Floating-Point Exception Trapping . . . . .	215
<b>Optimizing XL Fortran Programs . . . . .</b>	<b>217</b>
The Philosophy of XL Fortran Optimizations . . . . .	217
Choosing an Optimization Level . . . . .	219
Optimization level -O2 . . . . .	219
Optimization level -O3 . . . . .	220
Getting the most out of -O2 and -O3 . . . . .	220
The -O4 and -O5 Options . . . . .	221
Optimizing for a Target Machine or Class of Machines . . . . .	221
Getting the most out of target machine options . . . . .	222
Optimizing Floating-Point Calculations . . . . .	222
High-order transformations (-qhot) . . . . .	222
Getting the most out of -qhot . . . . .	223
Optimizing Loops and Array Language . . . . .	223
Profile-directed feedback (PDF) . . . . .	226
Optimizing Conditional Branching . . . . .	226
Interprocedural analysis (-qipa) . . . . .	226
Getting the most from -qipa . . . . .	227
Optimizing Subprogram Calls . . . . .	228
Finding the Right Level of Inlining . . . . .	228
Other Program Behavior Options . . . . .	229
Other performance options . . . . .	230
Debugging Optimized Code . . . . .	230
Different Results in Optimized Programs . . . . .	231
Compiler-friendly programming . . . . .	232
<b>Implementation Details of XL Fortran Input/Output . . . . .</b>	<b>233</b>
Implementation Details of File Formats . . . . .	233
File Names . . . . .	234
Preconnected and Implicitly Connected Files . . . . .	234
File Positioning . . . . .	235
I/O Redirection . . . . .	236
How XLF I/O Interacts with Pipes, Special Files, and Links . . . . .	236
Default Record Lengths . . . . .	237
File Permissions . . . . .	237
Selecting Error Messages and Recovery Actions . . . . .	237
Flushing I/O Buffers . . . . .	238
Choosing Locations and Names for Input/Output Files . . . . .	238
Naming Files That Are Connected with No Explicit Name . . . . .	238
Naming Scratch Files . . . . .	239
<b>Interlanguage Calls . . . . .</b>	<b>241</b>
Conventions for XL Fortran External Names . . . . .	241
Mixed-Language Input and Output . . . . .	242
Mixing Fortran and C++ . . . . .	242
Making Calls to C Functions Work . . . . .	244
Passing Data From One Language to Another . . . . .	245

Passing Arguments Between Languages . . . . .	245
Passing Global Variables Between Languages . . . . .	246
Passing Character Types Between Languages . . . . .	246
Passing Arrays Between Languages . . . . .	247
Passing Pointers Between Languages . . . . .	248
Passing Arguments By Reference or By Value . . . . .	248
Passing Complex Values to/from gcc . . . . .	250
Returning Values from Fortran Functions . . . . .	250
Arguments with the OPTIONAL Attribute . . . . .	250
Arguments with the INTENT Attribute . . . . .	250
Type Encoding and Checking . . . . .	251
Assembler-Level Subroutine Linkage Conventions . . . . .	251
The Stack . . . . .	252
The Link Area . . . . .	253
The Input Parameter Area . . . . .	253
The Register Save Area . . . . .	254
The Local Stack Area . . . . .	254
The Output Parameter Area . . . . .	254
Linkage Convention for Argument Passing . . . . .	254
Argument Passing Rules (by Value) . . . . .	255
Order of Arguments in Argument List . . . . .	256
Linkage Convention for Function Calls . . . . .	256
Pointers to Functions . . . . .	257
Function Values . . . . .	257
The Stack Floor . . . . .	257
Stack Overflow . . . . .	258
Prolog and Epilog . . . . .	258
Stack Size Limit . . . . .	258
<b>Problem Determination and Debugging . . . . .</b>	<b>259</b>
Understanding XL Fortran Error Messages . . . . .	259
Error Severity . . . . .	259
Compiler Return Code . . . . .	260
Run-Time Return Code . . . . .	260
Understanding XL Fortran Messages . . . . .	260
Limiting the Number of Compile-Time Messages . . . . .	261
Selecting the Language for Messages . . . . .	261
Fixing Installation or System Environment Problems . . . . .	262
Fixing Compile-Time Problems . . . . .	262
Duplicating Extensions from Other Systems . . . . .	262
Isolating Problems with Individual Compilation Units . . . . .	262
Running out of Machine Resources . . . . .	263
Fixing Link-Time Problems . . . . .	263
Fixing Run-Time Problems . . . . .	263
Duplicating Extensions from Other Systems . . . . .	263
Mismatched Sizes or Types for Arguments . . . . .	264
Working around Problems when Optimizing Input/Output Errors . . . . .	264
Tracebacks and Core Dumps . . . . .	264
Stack Size Overflows . . . . .	265
Debugging a Fortran 90 or Fortran 95 Program . . . . .	265
<b>Understanding XL Fortran Compiler Listings . . . . .</b>	<b>267</b>
Header Section . . . . .	267
Options Section . . . . .	267

Source Section . . . . .	268
Error Messages . . . . .	268
Transformation Report Section . . . . .	269
Attribute and Cross-Reference Section . . . . .	270
Object Section . . . . .	271
File Table Section . . . . .	271
Compilation Unit Epilogue Section . . . . .	271
Compilation Epilogue Section . . . . .	271

**Software Development Topics . . . 273**

**Porting Programs to XL Fortran . . . 275**

Outline of the Porting Process . . . . .	275
Portability of Directives . . . . .	275
Common Industry Extensions That XL Fortran Supports . . . . .	276
Mixing Data Types in Statements . . . . .	276
Date and Time Routines . . . . .	276
Other libc Routines . . . . .	276
Changing the Default Sizes of Data Types . . . . .	277
Name Conflicts Between Your Procedures and XL Fortran Intrinsic Procedures . . . . .	277
Reproducing Results from Other Systems . . . . .	277
Finding Nonstandard Extensions . . . . .	277

**Appendix A. Sample Fortran  
Programs . . . . . 279**

Example 1 - XL Fortran Source File . . . . .	279
Execution Results . . . . .	279
Example 2 - Valid C Routine Source File . . . . .	280

**Appendix B. XL Fortran Technical  
Information . . . . . 283**

The Compiler Phases . . . . .	283
External Names in XL Fortran Libraries . . . . .	283
The XL Fortran Run-Time Environment . . . . .	283
External Names in the Run-Time Environment . . . . .	284
Implementation Details for -qautodbl Promotion and Padding . . . . .	284
Terminology . . . . .	284
Examples of Storage Relationships for -qautodbl Suboptions . . . . .	286

**Appendix C. XL Fortran Internal Limits 291**

**Notices . . . . . 293**

Programming Interface Information . . . . .	295
Trademarks and Service Marks . . . . .	295

**Glossary . . . . . 297**

**INDEX . . . . . 301**

---

## Figures

1. Main Fortran Program That Calls C++ (main1.f) . . . . .	243	6. Storage Relationships with -qautodbl=dbl	287
2. C++ Wrapper Functions for Calling C++ (cfun.C) . . . . .	243	7. Storage Relationships with -qautobl=dbl4	288
3. C++ Code Called from Fortran (cplus.h)	244	8. Storage Relationships with -qautodbl=dbl8	288
4. Storage Mapping of Parm Area On the Stack	256	9. Storage Relationships with -qautodbl=dblpad4. . . . .	289
5. Storage Relationships without the -qautodbl Option. . . . .	286	10. Storage Relationships with -qautodbl=dblpad8. . . . .	289
		11. Storage Relationships with -qautodbl=dblpad	290



---

## Using XL Fortran Features

This section describes the features of the XL Fortran compiler and XL Fortran programs.

- “Introduction” on page 3
- “Overview of XL Fortran Features” on page 9
- “Setting Up and Customizing XL Fortran” on page 13
- “Editing, Compiling, Linking, and Running XL Fortran Programs” on page 23
- “XL Fortran Compiler-Option Reference” on page 43
- “XL Fortran Floating-Point Processing” on page 201
- “Optimizing XL Fortran Programs” on page 217
- “Implementation Details of XL Fortran Input/Output” on page 233
- “Interlanguage Calls” on page 241
- “Problem Determination and Debugging” on page 259
- “Understanding XL Fortran Compiler Listings” on page 267



---

## Introduction

This book describes Version 8.1 of IBM® XL Fortran Advanced Edition for Mac OS X and explains how to compile, link, and run programs that are written in the Fortran language.

---

## How to Use This Book

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

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

This book is not the place to find help on:

**Installation,**

which is covered in the *XL Fortran Advanced Edition for Mac OS X Installation Guide*.

**Writing Fortran programs,**

which is covered in the *XL Fortran Advanced Edition for Mac OS X Language Reference*.

The first part of this book 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 13.

**Create customized compiler defaults,**

read "Customizing the Configuration File" on page 15.

**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 43.

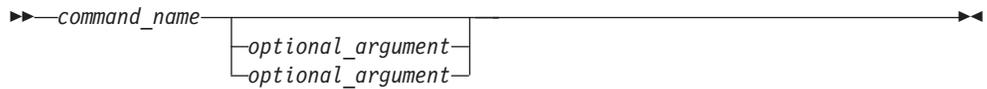
**Look up a particular option by name,**

scan alphabetically through "Detailed Descriptions of the XL Fortran Compiler Options" on page 62.

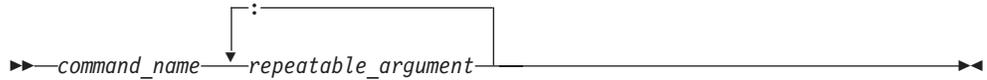
**Port a program to XL Fortran,**

read "Options for Compatibility" on page 53 to see what options you may need; then read "Porting Programs to XL Fortran" on page 275 for other porting information.



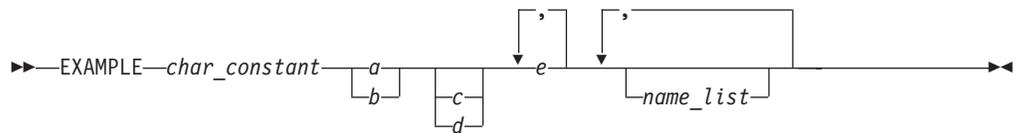


- An arrow returning to the left above the main line (a "repeat arrow") indicates an item that can be repeated and the separator character if it is other than a blank:



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 Book

- The examples in this book 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 book use the `xlf90`, `xlf90_r`, `xlf95`, `xlf95_r`, `xlf`, `xlf_r`, `f77`, and `fort77` 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 23.
- Some sample programs from this book and some other programs that illustrate ideas presented in this book are in the directory `/opt/ibmcmp/xlf/8.1/samples`.

## Notes on the Path Names in This Book

The path names shown in this book 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/8.1
```

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/8.1
```

## Notes on the Terminology in This Book

Some of the terminology in this book 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*.

---

## Related Documentation

You can refer to the following publications for additional information:

### XL Fortran and Operating System Publications

- *IBM XL Fortran Advanced Edition for Mac OS X Language Reference* describes the XL Fortran programming language.
- XL Fortran supplies brief installation instructions in the *XL Fortran Advanced Edition for Mac OS X Installation Guide* that explain how the general-purpose installation procedures apply to this licensed program.
- The Apple Help Center provides operating-system specific and other information.

### Standards Documents

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

- *American National Standard Programming Language Fortran 90*, ANSI X3.198-1992 (referred to in this book by its informal name, Fortran 90).
- *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 book by its informal name, Fortran 95).
- *American National Standard Programming Language FORTRAN*, ANSI X3.9-1978.
- *ANSI/IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Std 754-1985.
- *Federal (USA) Information Processing Standards Publication Fortran*, FIPS PUB 69-1.
- *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.



---

## 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 8.1 compiler is supported on the Mac OS X Versions 10.2 and 10.3 operating system.

The compiler, its generated object programs, and run-time library will run on Apple Power Mac G5 or G4 systems with the required software and disk space.

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. However, the compiler Scheduler models only the G5 architecture and does not exploit the G4 architecture optimally.

---

## 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 Advanced Edition for Mac OS X 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 book, 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<sup>®</sup> 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/8.1/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/ibmcmp/xlf/8.1/xlf.cfg` instead of creating many separate aliases or shell scripts.

For information on:

- The configuration file, see “Customizing the Configuration File” on page 15
- The invocation commands, see “Compiling XL Fortran Programs” on page 23

- The compiler options, see “Summary of the XL Fortran Compiler Options” on page 43 and “Detailed Descriptions of the XL Fortran Compiler Options” on page 62
- Compiler return codes, see “Understanding XL Fortran Messages” on page 260

---

## 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 51 and “Understanding XL Fortran Compiler Listings” on page 267.

---

## Symbolic Debugger Support

You can use **gdb** and other symbolic debuggers that support **gdb** style of debug information 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 217 is a road map to these features.

---

## Online Documentation

The XL Fortran books are available online in **HTML** and Portable Document Format (**PDF**) formats. The documentation is located in the following directories:

- `/opt/ibmcmp/xf/8.1/doc/en_US/html`
- `/opt/ibmcmp/xf/8.1/doc/en_US/pdf`

The HTML version of the documentation is searchable as part of the Apple Help Center. To display the HTML files, select Help from the Task bar. Then select Mac Help > IBM XL Fortran Compiler.

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.



---

## 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 books 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 any *Memo to Users* 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.
2. Read the file called `/opt/ibmcmp/xlf/8.1/doc/en_US/README.FIRST`, and follow any directions it gives. It contains information that you should know and possibly distribute to other people who use XL Fortran.

---

### 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 book, most examples of shell commands use **Bash** notation instead of repeating the syntax for all shells.

Part of the compilation environment are the search paths for special files such as libraries and include files. The following system variables are used by the compiler.

#### **DYLD\_LIBRARY\_PATH**

Specifies the directory path for dynamically loaded libraries. Used by the system linker at link time and at run time.

#### **MANPATH**

Specifies the search path for system and third-party software man pages.

## **PATH**

Specifies the directory path for the executable files of the compiler.

## **PDFDIR**

Specifies the directory in which the profile data file is created. The default value is unset, and the compiler places the profile data file in the current working directory. Setting this variable to an absolute path is recommended for profile-directed feedback.

## **TMPDIR**

Specifies the directory in which temporary files are created. The default location may be inadequate at high levels of optimization, where temporary files can require significant amounts of disk space.

## **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 239 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 `XLUNIT_unit` to file names. See “Naming Files That Are Connected with No Explicit Name” on page 238 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/xlf/8.1/xlf.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/ibmcmp/xlf/8.1/bin/xlf95 /Users/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:

<b>use</b>	The named and local stanzas provide the values for attributes. For single-valued attributes, values in the <b>use</b> attribute apply if there is no value in the local, or default, stanza. For comma-separated lists, the values from the <b>use</b> attribute are added to the values from the local stanza. You can only use a single level of the <b>use</b> attribute. Do not specify a <b>use</b> attribute that names a stanza with another <b>use</b> attribute.
<b>crt</b>	When invoked, 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.
<b>mcrt</b>	Same as for <b>crt</b> , but the object file contains profiling code for the <b>-p</b> option.
<b>gcr</b>	Same as <b>crt</b> , but the object file contains profiling code for the <b>-pg</b> option.
<b>gcc_libs</b>	When invoked, the linker options to specify the path to the GCC libraries and to link the GCC library.

<b>gcc_path</b>	Specifies the path to the tool chain.
<b>cpp</b>	The absolute path name of the C preprocessor, which is automatically called for files ending with a specific suffix (usually <b>.F</b> ).
<b>xlf</b>	The absolute path name of the main compiler executable file. The compiler commands are driver programs that execute this file.
<b>code</b>	The absolute path name of the optimizing code generator.
<b>xlfopt</b>	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.
<b>as</b>	The absolute path name of the assembler.
<b>asopt</b>	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 <b>-W</b> compiler option.
<b>ld</b>	The absolute path name of the linker.
<b>ldopt</b>	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 <b>-W</b> compiler option. However, most unrecognized options are passed to the linker anyway.
<b>options</b>	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.
<b>cppoptions</b>	A string of options that are separated by commas, to be processed by <b>cpp</b> as if you entered them on the command line before any other option. This attribute is needed because some <b>cpp</b> options are usually required to produce output that can be compiled by XL Fortran. The default options are <b>-C</b> (which preserves any C-style comments in the output) and <b>-P</b> (which prevents <b>#line</b> directives from being generated).
<b>fsuffix</b>	The allowed suffix for Fortran source files. The default is <b>f</b> . The compiler requires that all source files in a single compilation have the same suffix. Therefore, to compile files with other suffixes, such as <b>f95</b> , you must change this attribute in the configuration file or use the <b>-qsuffix</b> compiler option. For more information on <b>-qsuffix</b> , see “-qsuffix Option” on page 175.

<b>cppsuffix</b>	The suffix that indicates a file must be preprocessed by the C preprocessor ( <b>cpp</b> ) before being compiled by XL Fortran. The default is <b>F</b> .
<b>osuffix</b>	The suffix used to recognize object files that are specified as input files. The default is <b>o</b> .
<b>ssuffix</b>	The suffix used to recognize assembler files that are specified as input files. The default is <b>s</b> .
<b>libraries</b>	<b>-l</b> options, which are separated by commas, that specify the libraries used to link all programs.
<b>hot</b>	Absolute path name of the program that does array language optimizations when you specify the <b>-qhot</b> or <b>-qipa</b> options.
<b>ipa</b>	Absolute path name of the program that performs the following: <ul style="list-style-type: none"> <li>• interprocedural optimizations, when you specify the <b>-qipa</b> option</li> <li>• loop optimizations, when you specify the <b>-qhot</b> option</li> </ul>
<b>bolt</b>	Absolute path name of the binder (fast linker).
<b>defaultmsg</b>	Absolute path name of the default message files.
<b>include_32</b>	Indicates the search path that is used for the compiler supplied header and <b>.mod</b> files.

## What a Configuration File Looks Like

The following is an example of a configuration file:

```
xlf95:    use          = DEFLT
         libraries    = -lxlf90,-lxlopt,-lxlomp_ser,-lxl,-lxlfmath
         gcc_libs     = -lm,-lc,-lgcc,-lSystem
         options      = -qfree=f90

* Alias for standard Fortran compiler
f95:     use          = DEFLT
         libraries    = -lxlf90,-lxlopt,-lxlomp_ser,-lxl,-lxlfmath
         gcc_libs     = -lm,-lc,-lgcc,-lSystem
         options      = -qfree=f90
         fsuffix      = f95

* Fortran 90 compiler
xlf90:   use          = DEFLT
         libraries    = -lxlf90,-lxlopt,-lxlomp_ser,-lxl,-lxlfmath
         gcc_libs     = -lm,-lc,-lgcc,-lSystem
         options      = -qxlf90=noautodealloc:nosignedzero,-qfree=f90

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

* Original Fortran compiler
xlf:     use          = DEFLT
         libraries    = -lxlf90,-lxlopt,-lxlomp_ser,-lxl,-lxlfmath
         gcc_libs     = -lm,-lc,-lgcc,-lSystem
         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    = -lxl f90,-l xlopt,-l xlo mp_ser,-l xl,-l xlfmath
         gcc_libs     = -lm,-lc,-lgcc,-lSystem
         options      = -qnozerosize,-qsave,-qalias=intptr,-qposition=appendold,
                       -qxl f90=noautodealloc:nosignedzero,-qxl f77=intarg:intxor
                       :persistent:noleadzero:gedit77:noblankpad:oldboz:softeof

* Alias for original Fortran compiler, used for XPG4 compliance
fort77:  use          = DEFLT
         libraries    = -lxl f90,-l xlopt,-l xlo mp_ser,-l xl,-l xlfmath
         gcc_libs     = -lm,-lc,-lgcc,-lSystem
         options      = -qnozerosize,-qsave,-qalias=intptr,-qposition=appendold,
                       -qxl f90=noautodealloc:nosignedzero,-qxl f77=intarg:intxor
                       :persistent:noleadzero:gedit77:noblankpad:oldboz:softeof

* xlf with links to thread-safe components
xlf_r:   use          = DEFLT
         libraries    = -lxl f90_r,-l xlopt,-l xlo mp_ser,-l xl,-l xlfmath
         smplibraries = -lxl f90_r,-l xlopt,-l xlsmp,-l xl,-l xlfmath
         gcc_libs     = -lpthread,-lm,-lc,-lgcc,-lSystem
         options      = -qthreaded,-qnozerosize,-qsave,-qalias=intptr,
                       -qposition=appendold,-qxl f90=noautodealloc:nosignedzero,
                       -qxl f77=intarg:intxor:persistent:noleadzero:gedit77
                       :noblankpad:oldboz:softeof

* xlf90 with links to thread-safe components
xlf90_r: use          = DEFLT
         libraries    = -lxl f90_r,-l xlopt,-l xlo mp_ser,-l xl,-l xlfmath
         smplibraries = -lxl f90_r,-l xlopt,-l xlsmp,-l xl,-l xlfmath
         gcc_libs     = -lpthread,-lm,-lc,-lgcc,-lSystem
         options      = -qxl f90=noautodealloc:nosignedzero,-qfree=f90,-qthreaded

* xlf95 with links to thread-safe components
xlf95_r: use          = DEFLT
         libraries    = -lxl f90_r,-l xlopt,-l xlo mp_ser,-l xl,-l xlfmath
         smplibraries = -lxl f90_r,-l xlopt,-l xlsmp,-l xl,-l xlfmath
         gcc_libs     = -lpthread,-lm,-lc,-lgcc,-lSystem
         options      = -qfree=f90,-qthreaded

* Common definitions
DEFLT:  xlf          = /opt/ibmcmp/xlf/8.1/exe/xlfentry
        crt          = /usr/lib/crt1.o
        mcrt         = /usr/lib/gcrt1.o
        gcrt         = /usr/lib/gcrt1.o
        include_32   = -I/opt/ibmcmp/xlf/8.1/include
        dis          = /opt/ibmcmp/xlf/8.1/exe/dis
        code         = /opt/ibmcmp/xlf/8.1/exe/xlfcode
        hot          = /opt/ibmcmp/xlf/8.1/exe/xlfhot
        ipa          = /opt/ibmcmp/xlf/8.1/exe/ipa
        bolt         = /opt/ibmcmp/xlf/8.1/exe/bolt
        defaultmsg   = /opt/ibmcmp/xlf/8.1/msg/en_US
        as           = /usr/bin/as
        ld           = /usr/bin/ld
        cppoptions   = -C
        cpp          = /opt/ibmcmp/xlf/8.1/exe/cpp
        dynlib       = -dynamic
        libdirs      = -L/opt/ibmcmp/xlsmp/1.3/lib,-L/opt
                       /ibmcmp/xlf/8.1/lib,-L/opt/ibmcmp/xlsmp
                       /1.3/../../lib,-L/opt/ibmcmp/xlf/8.1/../../lib
        gcc_path     = /usr
        gcc_libdirs  = -L/usr/lib/gcc/darwin/3.3,-L/usr/lib/gcc/darwin,-
                       L/usr/libexec/gcc/darwin/ppc/3.3/../../
        options      =
        modes_config = 32

```

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

**Related Information:** You can use the “-F Option” on page 70 to select a different configuration file, a specific stanza in the configuration file, or both.

---

## Using XL Fortran with Xcode and Project Builder

The XL Fortran compiler is a command-line utility similar to its GCC counterpart. In addition to having a command-line interface, XL Fortran can be used in the Apple integrated development environments (IDEs) Xcode and Project Builder.

New projects created by Xcode use the native build system. On the other hand, Project Builder uses the `jam` utility, which is similar to the `make` utility commonly used on other UNIX platforms. The XL compilers do not support a jam-based build from within Xcode. You can either upgrade an existing Project Builder project to a native Xcode project, or you can use the XL compilers with Project Builder to build the project. It is recommended that further development concentrates on using the Xcode IDE for native builds and moves away from jam-based projects.

### Configure the Xcode IDE

The Xcode IDE uses specification files to describe the compilers and associated utilities that are used in a build. During the XL compiler installation, the compiler specification file `XLF.8.1.pbcompspec` is copied into the `/Library/Application Support/Apple/Developer Tools/Specifications` directory. This file does not change your existing Xcode settings. You can continue to use `gcc` as your build compiler, or you can use XL Fortran, as described in the next section.

### Setting Up XL Fortran to Use Xcode Projects

This section describes how to build projects using XL Fortran.

The following steps describe how to modify the Xcode IDE settings to use the `gxlfc` utility to compile Fortran source files. This utility is used as a bridge between Xcode and XL Fortran. Most XL Fortran options are available in the **Build Settings** pane. It is also possible to add compiler and linker options explicitly.

Any XL Fortran compiler option that has no corresponding `gxlfc` equivalent must be prefixed with `-Wx` when specified for `gxlfc`. For example, to use the `-qhot` compiler option, specify `-Wx,-qhot`.

The following steps describe how to set up an Xcode project to use the XL Fortran compiler:

1. Start the Xcode IDE
2. Create a new Empty project or open a sample project included. Ensure you have write permission to the project's directory.

If you are using a sample project, continue at step 3 on page 20.

If you created an Empty project, first do the following:

- a. Create a new file with `.f` as the extension.
- b. Add your code to the file.
- c. Click the **Targets** tab of the project.

- d. Add a new **Tool** target.
3. Select the project in the **Groups & Files** list.
4. Click the **Info** button on the toolbar to display the **Info** window of the project.
5. Click the **Styles** tab to open the **Styles** pane of the project.
6. Clear the **Zero Link**, **Prebinding**, and **Fix & Continue** options if they are selected. XL Fortran does not support them.
7. Close the **Info** window of the project.
8. Select the target that is being configured from the **Groups & Files** list.
9. Click the **Info** button on the toolbar to display the **Info** window of the target.
10. Click the **Rules** tab.
11. Click the plus (+) button at the bottom left of the **Rules** pane to add a new rule.
12. From the **Process** list, specify the type of source files for the new rule. To use the XL Fortran compiler, select **FORTRAN source files**.
13. From the **Using** list, select **IBM XL Fortran Version 8.1**.
14. Click the **Build** tab.
15. Clear the **Precompile prefix header** if it is selected. XL Fortran does not support it.
16. Add compiler options:
  - Specify XL Fortran compiler options in the Build pane of the target. Note that only those that can be translated by **gxlfc** can be specified this way. For those that cannot be translated, specify them as XL Fortran compiler options as described in the next list item.
  - Specify XL Fortran compiler options by adding them to **Current Settings/OTHER\_CFLAGS**, using the format **-Wx,-ibm\_option**.
  - Specify XL Fortran linking options in **Standard Build Settings/Linking/Other linker flags**, using the format **-Wx,-ibm\_option**.
17. Close the **Info** window of the target.
18. Build the project.

### Hints and Tips in Using XL Fortran with Xcode

You need to do the preceding setup only once for a project. Here are some additional considerations:

- Most of the compiler and linker options are provided under the **Build** pane of the project settings, but they can also be added as outlined in step 16, above.
- A project has both **Target** and **Project** settings. These are in different locations, so check in both places when setting options for a project.

## Using XL Fortran with Project Builder

You can redirect the Project Builder IDE to use **gxlfc** to translate a GNU compiler option into an XL Fortran equivalent. The **xlfc\_pb** script provides the necessary interface to invoke **gxlfc**.

Any XL Fortran compiler option that has no corresponding **gxlfc** equivalent must be prefixed with **-Wx,**. For example, to use the **-qhot** compiler option, specify **-Wx,-qhot**. (See “Setting the CC Variable” on page 21.)

The following steps describe how to use Project Builder with the XL compilers:

1. Start Project Builder
2. Create an Empty Project or open an existing project.
  - If you are using an existing project, continue at step 3 on page 21.
  - If you created an Empty project, first do the following:
    - a. Create a new file with **.f** as the extension.

- b. Add your code to the file.
  - c. Click the **Targets** tab of the project.
  - d. Add a new **Tool** target.
3. Edit the build settings for the target as follows:
    - Set the **CC** variable. See “Setting the CC Variable.”
    - Specify XL Fortran compiler options in the format **-Wx,-ibm\_option**.
  4. Uncheck the **Prebinding** linker option.
  5. Click the **files** tab of the project.
  6. Click the checkbox next to the source file.
  7. Build the project.

### Hints and Tips in Using XL Fortran with Project Builder

The considerations in “Hints and Tips in Using XL Fortran with Xcode” on page 20 also apply to Project Builder.

## Setting the CC Variable

The `xlf_pb` script is used to provide the interface to Project Builder. It does the necessary setup and then runs `gxlfc`. To direct Project Builder to use this script, you must set the **CC** variable if the project contains Fortran source files. You need to do this only once for a project.

To set the **CC** variable, do the following:

1. Open an existing project or create a new one.
2. Click the **Targets** tab to display a contents list.
3. In the contents list, under **Targets**, select the name of the target you are working on. The Target Summary pane appears.
4. From the navigation tab on the left, select **Settings > Expert View**. A list of **Build Settings** appears.
5. Add an entry to the **Build Settings** by clicking the plus sign (+) at the bottom left of the list.
6. In the **Name** field, rename the new setting **CC**. In the **Value** field, enter the full path to the `xlf_pb` script. The script is located in the `$xlfPath/exe/` directory. For example, if you used the default installation location, the value you should enter is:

```
/opt/ibmcomp/xlf/8.1/exe/xlf_pb
```



---

## 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 Mac OS X 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 13.
2. To learn more about writing Fortran programs, refer to the *XL Fortran Advanced Edition for Mac OS X Language Reference*.

---

## Editing XL Fortran Source Files

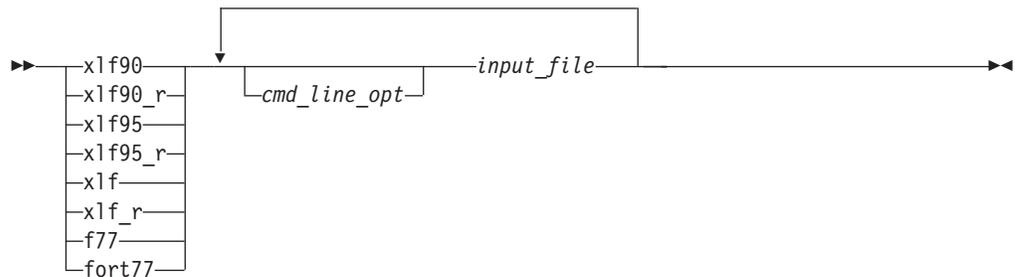
To create Fortran source programs, you can use any of the available text editors, such as **vi**, **emacs**, or **TextEdit**. Source programs must have a suffix of **.f** unless the **fsuffix** attribute in the configuration file specifies a different suffix or the **-qsuffix** compiler option is used. 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 Advanced Edition for Mac OS X Language Reference*.

---

## Compiling XL Fortran Programs

To compile a source program, use one of the **xl f90**, **xl f90\_r**, **xl f95**, **xl f95\_r**, **xl f**, **xl f\_r**, **f77**, or **fort77** 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/ibmcmp/xlf/8.1/xlf.cfg`).

The invocation command performs the necessary steps to compile the Fortran source files and link the object files and libraries into an executable program. In particular, the **xl f\_r**, **xl f90\_r**, and **xl f95\_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
<b>xlf90 f90</b>	/opt/ibmcmp/xlf/8.1/bin	Fortran 90	libxlf90.dylib
<b>xlf90_r</b>	/opt/ibmcmp/xlf/8.1/bin	Threadsafe Fortran 90	libxlf90_r.dylib
<b>xlf95 f95</b>	/opt/ibmcmp/xlf/8.1/bin	Fortran 95	libxlf90.dylib
<b>xlf95_r</b>	/opt/ibmcmp/xlf/8.1/bin	Threadsafe Fortran 95	libxlf90_r.dylib
<b>xlf</b>	/opt/ibmcmp/xlf/8.1/bin	FORTTRAN 77	libxlf90.dylib
<b>xlf_r</b>	/opt/ibmcmp/xlf/8.1/bin	Threadsafe FORTRAN 77	libxlf90_r.dylib
<b>f77 or fort77</b>	/opt/ibmcmp/xlf/8.1/bin	FORTTRAN 77	libxlf90.dylib

The invocation commands have the following implications for directive triggers:

- For **f77**, **fort77**, **xlf**, **xlf90**, and **xlf95**, the directive trigger is **IBM\*** by default.
- For all other commands, the directive triggers are **IBM\*** and **IBMT** by default.

XL Fortran provides the library **libxlf90\_r.dylib**, in addition to **libxlf90\_t.dylib**. The library **libxlf90\_r.dylib** is a superset of **libxlf90\_t.dylib**. The file **xlf.cfg** is set up to link to **libxlf90\_r.dylib** automatically when you use the **xlf90\_r**, **xlf95\_r**, and **xlf\_r** commands.

**libxlf90\_t.dylib** is a partial thread-support run-time library. It will be installed as **/opt/ibmcmp/lib/ibxlf90\_t.dylib** 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.dylib**, you can use **libxlf90\_t.dylib** 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 **libxlf90\_r.dylib**, **-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 **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 **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. **xlf90**, **xlf90\_r**, **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
```

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

```
export XLF RTEOPTS="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**.

## 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. 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**.

**Note:** File names on Mac OS X are case-insensitive. The files **test.f** and **test.F** are considered to have the same name and cannot be in the same directory. The XL Fortran compiler, however, recognizes case-sensitive file names.

**Related Information:** See “Passing Fortran Files through the C Preprocessor” on page 31.

The **fsuffix** and **cppsuffix** attributes in “Customizing the Configuration File” on page 15 and the “-qsuffix Option” on page 175 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 libraries in the product and system library directories. It then produces a single executable output file.

**Related Information:** See “Options That Control Linking” on page 59 and “Linking XL Fortran Programs” on page 32.

The **osuffix** attribute, which is described in “Customizing the Configuration File” on page 15 and the “-qsuffix Option” on page 175, 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 15 and the “-qsuffix Option” on page 175, lets you select a different suffix.

#### **Shared Object or Library Files (.dylib 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/ibmcmp/xlf/8.1/xlf.cfg**.

**Related Information:** See “Customizing the Configuration File” on page 15 and “-F Option” on page 70 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 72 and “-qmoddir Option” on page 147.

#### **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 153.

## **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 79 for information on selecting a different name and “-c Option” on page 67 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 67 and “Linking XL Fortran Programs” on page 32. For information on renaming the object file, see “`-o` Option” on page 79.

**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 51.

**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 147.

**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 31 and “`-d` Option” on page 69.

**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 153.

## 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 <b>@PROCESS</b> directive. (XL Fortran also has comment directives, such as <b>SOURCEFORM</b> , 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 62 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
```

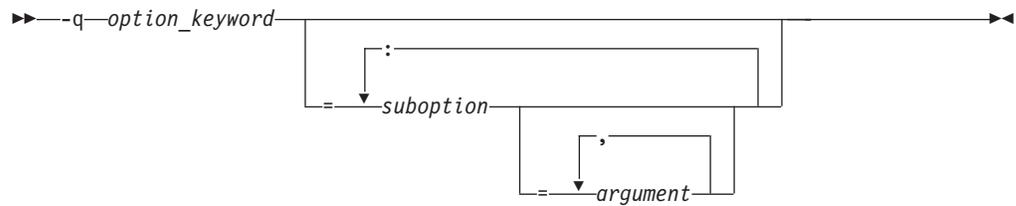
(There are some exceptions, such as **-pg**, which is a single option and not the same as **-p -g**.)

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 **l** 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/ibmcomp/xlf/8.1/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/ibmcomp/xlf/8.1/xlf.cfg`:

```
asopt = "W"  
ldopt = "w"
```

and issue this command:

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

the file `uses_many_symbols.f` is compiled with **-w** (short form of **-qflag=e:e**) and the file `produces_warnings.s` is assembled with the options **-W** and **-g** (suppress warnings and produce debug information), and the object files are linked with the options **-w** and **-s** (produce warnings, strip final executable file).

**Related Information:** See “**-W** Option” on page 197 and “Customizing the Configuration File” on page 15.

## Compiling for PowerPC Systems

You can use **-qarch** and **-qtune** to target your program to instruct the compiler to generate code specific to Power Mac G5 or general PowerPC® architectures. 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 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 86 and also **-O4** and **-O5** under the “**-O** Option” on page 77.

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. If you are also using one of the optimizing preprocessors, **cpp** is called before the other preprocessor.

To call **cpp** for a particular file, use a file suffix of **.F**. File names on Mac OS X are case-insensitive. The files **test.f** and **test.F** are considered to have the same name and cannot be in the same directory. The XL Fortran compiler, however, recognizes case-sensitive file names.

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.

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 69, “-t Option” on page 192, “-W Option” on page 197, and “Customizing the Configuration File” on page 15.

## 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 **MACVX**, you can define that symbol to **cpp** by compiling with a command like:

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

## 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/8.1/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 linker is executed automatically to produce an executable output file:

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

After linking, follow the instructions in “Running XL Fortran Programs” on page 34 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
xlf95 -c file2.f file3.f   # Or multiple object files
xlf95 file1.o file2.o file3.o # Link 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...> -K -r file.f # xlf95 passes all these options to ld
```

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

## Dynamic Linking

XL Fortran allows your programs to take advantage of the operating system facilities for dynamic 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 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.

## 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 **mtime\_** and **flush\_**.

- Link your programs dynamically, which is the default. Many naming conflicts only apply to statically linked programs.

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 **\_xlf**.
- 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 safe to use, you can use 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:

XL F-Provided Function Name	Common Block or Subprogram Name You Cannot Use
mclock	times
rand	irand

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 the archive versions of the Fortran run-time libraries (for example, **libxlf90.a**), 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. **libxlf90.dylib** is usually the only XL Fortran library needed. **libxlfpm\*.a** and **libxlfpad.a** 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 operating system on the execution system must be at either the same level or a more recent level than on the compilation system.

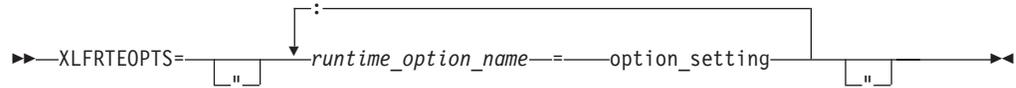
**Related information:** See “Dynamic Linking” on page 33.

### 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 the **XLFRTEOPTS** environment variable.

## The XLFRT\_OPTS Environment Variable

The **XLFRT\_OPTS** environment variable allows you to specify options that affect I/O, EOF error-handling, and the specification of random-number generators. You can declare **XLFRT\_OPTS** 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 **XLFRT\_OPTS** 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.

Changing the **XLFRT\_OPTS** environment variable during the execution of a program has no effect on the program.

The **SETRT\_OPTS** procedure (which is defined in *Service and Utility Procedures* in the *XL Fortran Advanced Edition for Mac OS X Language Reference*) accepts a single-string argument that contains the same name-value pairs as the **XLFRT\_OPTS** 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 **SETRT\_OPTS**. Only the settings that you specified in the procedure call are changed.

You can specify the following run-time options with the **XLFRT\_OPTS** environment variable or the **SETRT\_OPTS** 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:

<b>enable</b>	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.
<b>disable_preconn</b>	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.
<b>disable_all</b>	The Fortran run-time library does not maintain I/O buffers for any logical units. You should not specify the <b>buffering=disable_all</b> 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;
}
```

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.

**cnvrr={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 Advanced Edition for Mac OS X Language Reference*. For more information about **IOSTAT** values, see *Conditions and IOSTAT Values* in the *XL Fortran Advanced Edition for Mac OS X Language Reference*.

**cpu\_time\_type={usertime | system | alltime | total\_usertime | total\_system | 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.

**system**

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\_system**

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.

**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.

**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* equals the number of processors online.

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

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

<b>90std</b>	Specifies that the compiler should flag any extensions to the Fortran 90 standard I/O statements and formats as errors.
<b>95std</b>	Specifies that the compiler should flag any extensions to the Fortran 95 standard I/O statements and formats as errors.
<b>extended</b>	Specifies that the compiler should accept all extensions to the Fortran 90 standard and Fortran 95 standard I/O statements and formats.

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**
- **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
```

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'**)
- 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 either the **langlvl=95std** or the **langlvl=90std** setting.

**Related Information:** For more information about **NAMELIST** I/O, see *Namelist Formatting in the XL Fortran Advanced Edition for Mac OS X 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 **XLFSRATCH\_unit** to the name of the file you want to be associated with the specified unit number. See “Naming Scratch Files” on page 239 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 238 for examples.

**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
XLFRTOPTS=cnverr=yes:xrf_messages=no
export XLFRTOPTS

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

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.
```

---

## Other Environment Variables That Affect Run-Time Behavior

The **DYLD\_LIBRARY\_PATH** and **TMPDIR** environment variables have an effect at run time, as explained in “Correct Settings for Environment Variables” on page 13. They are not XL Fortran run-time options and cannot be set in **XLFRTOPTS**.

---

## 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.

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

- Fixed-point division by zero.
- Floating-point exceptions occur after 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 66, “-qflttrap Option” on page 117, and “-qsigtrap Option” on page 168.

See “Detecting and Trapping Floating-Point Exceptions” on page 209 for more details about these exceptions and “Controlling the Floating-Point Status and Control Register” on page 212 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 62.

---

### 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 book 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 25 and “Options That Specify the Locations of Output Files” on page 45.

Many of the options in “Options for Compatibility” on page 53 change the permitted input format slightly.

Table 2. Options That Control Input to the Compiler

Command-Line Option	@PROCESS Directive	Description	See Page
-Idir		<p>Adds a directory to the search path for include files and <b>.mod</b> files. If XL Fortran calls <b>cpp</b>, this option adds a directory to the search path for <b>#include</b> files. Before checking the default directories for include and <b>.mod</b> files, the compiler checks each directory in the search path. For include files, this path is only used if the file name in an <b>INCLUDE</b> line is not provided with an absolute path. For <b>#include</b> files, refer to the <b>cpp</b> documentation for the details of the <b>-I</b> option.</p> <p><b>Default:</b> The following directories are searched, in the following order:</p> <ol style="list-style-type: none"> <li>1. The current directory</li> <li>2. The directory where the source file is</li> <li>3. <b>/usr/include</b>.</li> </ol> <p>Also, <b>/opt/ibmcmp/xf/8.1/include</b> is searched; include and <b>.mod</b> files that are shipped with the compiler are located here.</p>	72
-qci=numbers	CI(numbers)	<p>Activates the specified <b>INCLUDE</b> lines.</p> <p><b>Default:</b> No default value.</p>	97
-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><b>Default:</b> <b>-qcr</b></p>	100
-qdirective [=directive_list]	DIRECTIVE [(directive_list)]	<p>Specifies sequences of characters, known as trigger constants, that identify comment lines as compiler comment directives.</p> <p><b>Default:</b> Comment lines beginning with <b>IBM*</b> are considered directives.</p>	105
-qnodirective [=directive_list]	NODIRECTIVE [(directive_list)]		

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

Command-Line Option	@PROCESS Directive	Description	See Page
-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. <b>Default:</b> -qfree=f90 for the xlf90, xlf90_r, xlf95, and xlf95_r commands and -qfixed=72 for the xlf, xlf_r, and f77/fort77 commands.	113
-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. <b>Default:</b> -qfree=f90 for the xlf90, xlf90_r, xlf95, and xlf95_r commands and -qfixed=72 for the xlf, xlf_r, and f77/fort77 commands.	119
-qmbcs -qnombs	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. <b>Default:</b> -qnombs	145
-qOBJect -qNOOBJect	OBJect NOOBJect	Specifies whether to produce an object file or to stop immediately after checking the syntax of the source files. <b>Default:</b> -qobject	150
-U -qmixed -qnomixed	MIXED NOMIXED	Makes the compiler sensitive to the case of letters in names. <b>Default:</b> -qnomixed	193
-qsuffix={suboptions}		Specifies the source-file suffix on the command line.	175

## 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 Mac OS X information for the **ld** command.

**Related Information:** See “XL Fortran Output Files” on page 26 and “Options That Control Input to the Compiler” on page 44.

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 <b>cpp</b> , instead of deleting them. <b>Default:</b> Temporary files produced by <b>cpp</b> are deleted.	69
-o <i>name*</i>		Specifies a name for the output object, executable, or assembler source file. <b>Default:</b> -o a.out	79
-qmoddir= <i>directory</i>		Specifies the location for any module ( <b>.mod</b> ) files that the compiler writes. <b>Default:</b> <b>.mod</b> files are placed in the current directory.	147

## 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 217.

Some of the options in “Options for Floating-Point Processing” on page 59 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). <b>Default:</b> <b>-qnooptimize</b>	77
-p -pg		Sets up the object file for profiling. <b>Default:</b> No profiling.	80
-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. <b>Default:</b> - <b>qalias=aryovrlp:nointptr:pteovrlp:std</b> for the <b>xlF90</b> , <b>xlF90_r</b> , <b>xlF95</b> and <b>xlF95_r</b> commands; <b>-qalias=aryovrlp:intptr:pteovrlp:std</b> for the <b>xlF</b> , <b>xlF_r</b> , <b>f77</b> , and <b>fort77</b> commands.	81

Table 4. Options for Performance Optimization (continued)

Command-Line Option	@PROCESS Directive	Description	See Page
-qalign={[no]4k   struct {=packed   natural   port}}	ALIGN( ([NO]4K   STRUCT{(packed   (natural)   (port)}))	Specifies the alignment of data objects in storage, which avoids performance problems with misaligned data. Both the <b>[no]4k</b> and <b>struct</b> options can be specified and are not mutually exclusive. The default setting is <b>-qalign=no4k:struct=natural</b> . The <b>[no]4K</b> option is useful primarily in combination with logical volume I/O and disk striping. <b>Default: -qalign= no4k:struct=natural</b>	84
-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. The choices are auto, g5, ppcv, and ppc970. <b>Default: -qarch=ppcv</b>	86
-qassert={ deps   nodeps   itercnt= <i>n</i> }		Provides information about the characteristics of the files that can help to fine-tune optimizations. <b>Default: -qassert= deps:itercnt=1024</b>	88
-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. <b>Default:</b> The compiler uses typical values based on the <b>-qtune</b> setting, the <b>-qarch</b> setting, or both settings.	92
-qcompact -qnocompact	COMPACT NOCOMPACT	Reduces optimizations that increase code size. <b>Default: -qnocompact</b>	99
-qhot[= <i>suboptions</i> ] -qnohot	HOT[= <i>suboptions</i> ] NOHOT	Determines whether to perform high-order transformations on loops and array language during optimization and whether to pad array dimensions and data objects to avoid cache misses. <b>Default: -qnohot</b>	122
-qipa[= <i>suboptions</i> ]   -qnoipa		Enhances <b>-O</b> optimization by doing detailed analysis across procedures (interprocedural analysis or IPA). <b>Default: -O</b> analyzes each subprogram separately, ruling out certain optimizations that apply across subprogram boundaries. Note that specifying <b>-O5</b> is equivalent to specifying <b>-O4</b> and <b>-qipa=level=2</b> .	130

Table 4. Options for Performance Optimization (continued)

Command-Line Option	@PROCESS Directive	Description	See Page
-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. <b>Default:</b> Optimizations use fixed assumptions about branch frequency and other statistics.	153
-qprefetch   -qnoprefetch		Indicates whether or not the prefetch instructions should be inserted automatically by the compiler. <b>Default:</b> <b>-qprefetch</b>	160
-qsmallstack -qnosmallstack		Specifies that the compiler will minimize stack usage where possible. <b>Default:</b> <b>-qnosmallstack</b>	169
-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. <b>Default:</b> 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.	172
-qstrictieemod -qnostrictieemod	STRICTIEEE- MOD NOSTRICTIEEE- MOD	Specifies whether the compiler will adhere to the Fortran 2000 IEEE arithmetic rules for the <b>ieee_arithmetic</b> and <b>ieee_exceptions</b> intrinsic modules. <b>Default:</b> <b>-qstrictieemod</b>	173
-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. <b>Default:</b> <b>-qnostrict_induction</b>	174
-qthreaded		Specifies that the compiler should generate thread-safe code. This is turned on by default for the <b>xlf_r</b> , <b>xlf90_r</b> , and <b>xlf95_r</b> commands.	178
-qtune= <i>implementation</i>		Tunes instruction selection, scheduling, and other implementation-dependent performance enhancements for a specific implementation of a hardware architecture. The following settings are valid: auto, g5, ppc970. <b>Default:</b> <b>-qtune=ppc970</b>	179
-qunroll [=auto   yes] -qnounroll		Specifies whether the compiler is allowed to automatically unroll <b>DO</b> loops. <b>Default:</b> <b>-qunroll=auto</b>	181

Table 4. Options for Performance Optimization (continued)

Command-Line Option	@PROCESS Directive	Description	See Page
-qunwind -qnounwind	UNWIND NOUNWIND	Specifies default behavior for saving and restoring from non-volatile registers during a procedure call. <b>Default: -qunwind</b>	182
-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. <b>Default: -qzerosize</b> for the <code>xlF90</code> , <code>xlF90_r</code> , <code>xlF95</code> , and <code>xlF95_r</code> commands and <b>-qnozerosize</b> for the <code>xlF</code> , <code>xlF_r</code> , <code>f77</code> , and <code>fort77</code> commands (meaning these commands cannot be used for programs that contain zero-sized objects).	191

## 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 259.

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 **-qflttrap**, 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 63, “-v Option” on page 195, and “-V Option” on page 196
- “-qalias Option” on page 81
- “-qci Option” on page 97
- “-qobject Option” on page 150
- “-qreport Option” on page 165
- “-qsource Option” on page 170

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 <b>SIGTRAP</b> signals at run time. <b>Default: -qnocheck</b>	66
-D -qdlines -qnodlines	DLINES NODLINES	Specifies whether fixed source form lines with a D in column 1 are compiled or treated as comments. <b>Default: -qnodlines</b>	68
-g -qdbg -qnodbg	DBG NODBG	Generates debug information for use by a symbolic debugger. <b>Default: -qnodbg</b>	71
-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 <b>SIGFPE</b> signal when the corresponding exception occurs. <b>Default: -qnofltrap</b>	117
-qfullpath -qnofullpath		Records the full, or absolute, path names of source and include files in object files compiled with debugging information (-g option). <b>Default:</b> The relative path names of source files are recorded in the object files.	120
-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". <b>Default: -qhalt=S</b>	121
-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 <b>-qinitauto</b> option to initialize <b>REAL</b> variables with a signaling NAN value and the <b>-qfltrap</b> option, it is possible to identify references to uninitialized <b>REAL</b> variables at run time. <b>Default: -qnoinitauto.</b> If you specify <b>-qinitauto</b> without a <i>hex_value</i> , the compiler initializes the value of each byte of automatic storage to zero.	125

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

Command-Line Option	@PROCESS Directive	Description	See Page
-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. <b>Default: -qlanglvl=extended</b>	136
-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.	166
-qsigtrap[= trap_handler]		Installs xl__trce or a predefined or user-written trap handler in a main program. <b>Default:</b> No trap handler installed; program core dumps when a <b>trap</b> instruction is executed.	168
-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. <b>Default: -qnoxlines</b>	188

## 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 49 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. <b>Default:</b> No progress messages are produced.	63
-qattr[=full] -qnoattr	ATTR[(FULL)] NOATTR	Specifies whether to produce the attribute component of the attribute and cross-reference section of the listing. <b>Default: -qnoattr</b>	89

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

Command-Line Option	@PROCESS Directive	Description	See Page
-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. <b>Default: -qflag=i:i</b>	114
-qlist -qnolist	LIST NOLIST	Specifies whether to produce the object section of the listing. <b>Default: -qnolist</b>	140
-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. <b>Default: -qnolistopt</b>	141
-qnoprint		Prevents the listing file from being created, regardless of the settings of other listing options. <b>Default:</b> Listing is produced if you specify any of -qattr, -qlist, -qlistopt, -qphsinfo, -qreport, -qsource, or -qxref.	148
-qphsinfo -qnophsinfo	PHSINFO NOPHSINFO	Determines whether timing information is displayed on the terminal for each compiler phase. <b>Default: -qnophsinfo</b>	156
-qreport[={ hotlist}...] -qnoreport	REPORT [({ HOTLIST}...)] NOREPORT	Determines whether to produce transformation reports showing how loops are optimized. <b>Default: -qnoreport</b>	165
-qsource -qnosource	SOURCE NOSOURCE	Determines whether to produce the source section of the listing. <b>Default: -qnosource</b>	170
-qsuppress [= <i>nnnn-mmm</i> [ <i>nnnn-mmm</i> ...]   cmpmsg]   -qnosuppress		Specifies which messages to suppress from the output stream.	176
-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. <b>Default: -qnoxref</b>	190
-v		Traces the progress of the compilation by displaying the name and parameters of each compiler component that is executed by the invocation command. <b>Default:</b> No progress messages are produced.	195

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

Command-Line Option	@PROCESS Directive	Description	See Page
-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. <b>Default:</b> No progress messages are produced.	196

## 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 275 discusses this subject in more detail. “Duplicating the Floating-Point Results of Other Systems” on page 209 explains how to use some of the options in “Options for Floating-Point Processing” on page 59 to achieve floating-point results compatible with other systems.

The **-qfree=ibm** form of the “-qfree Option” on page 119 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. <b>Default:</b> -qautodbl=none	90
-qcclines -qnocclines	CCLINES NOCCLINES	Determines whether the compiler recognizes conditional compilation lines. <b>Default:</b> -qnocclines.	94

Table 7. Options for Compatibility (continued)

Command-Line Option	@PROCESS Directive	Description	See Page
-qctyp <sub>lss</sub> [=( <sub>no</sub> )arg] -qnoctyp <sub>lss</sub>	CTYPLSS [[ <sub>NO</sub> ]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 <b>arg</b> specifies that Hollerith constants used as actual arguments will be treated as integer actual arguments. <b>Default: -qnoctyp<sub>lss</sub></b>	101
-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. <b>Default: -qnoddim</b>	104
-qdpc -qdpc=e -qnodpc	DPC DPC(E) NODPC	Increases the precision of real constants, for maximum accuracy when assigning real constants to <b>DOUBLE PRECISION</b> variables. This language extension might be needed when you are porting programs from other platforms. <b>Default: -qnodpc</b>	108
-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. <b>Default: -qescape</b>	109

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 <b>EXTERNAL</b> 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. <b>Default:</b> Names of intrinsic procedures override user-written procedure names when they are the same.	110
-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. <b>Default:</b> -qnoextname	111
-qintlog -qnointlog	INTLOG NOINTLOG	Specifies that you can mix integer and logical values in expressions and statements. <b>Default:</b> -qnointlog	127
-qintsize= <i>bytes</i>	INTSIZE( <i>bytes</i> )	Sets the size of default <b>INTEGER</b> and <b>LOGICAL</b> values. <b>Default:</b> -qintsize=4	128
-qlog4 -qnolog4	LOG4 NOLOG4	Specifies whether the result of a logical operation with logical operands is a <b>LOGICAL(4)</b> or is a <b>LOGICAL</b> with the maximum length of the operands. <b>Default:</b> -qnolog4	142

Table 7. Options for Compatibility (continued)

Command-Line Option	@PROCESS Directive	Description	See Page
-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. <b>Default: -qnonullterm</b>	149
-1 -qonetrip -qnoonetrip	ONETRIP NOONETRIP	Executes each <b>DO</b> loop in the compiled program at least once if its <b>DO</b> statement is executed, even if the iteration count is 0. <b>Default: -qnoonetrip</b>	64
-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. <b>Default: -qnoport</b>	158
-qposition= {appendold   appendunknown}	POSITION( {APPENDOLD   APPENDUNKNOWN})	Positions the file pointer at the end of the file when data is written after an <b>OPEN</b> statement with no <b>POSITION=</b> specifier and the corresponding <b>STATUS=</b> value ( <b>OLD</b> or <b>UNKNOWN</b> ) is specified. <b>Default:</b> Depends on the I/O specifiers in the <b>OPEN</b> statement and on the compiler invocation command: <b>-qposition=appendold</b> for the <b>xl</b> f, <b>xl</b> f_r, and <b>f77/fort77</b> commands and the defined Fortran 90 and Fortran 95 behaviors for the <b>xl</b> f90, <b>xl</b> f90_r, <b>xl</b> f95, and <b>xl</b> f95_r commands.	159

Table 7. Options for Compatibility (continued)

Command-Line Option	@PROCESS Directive	Description	See Page
-qqcount -qnoqcount	QCOUNT NOQCOUNT	Accepts the <b>Q</b> character-count edit descriptor ( <b>Q</b> ) as well as the extended-precision <b>Q</b> edit descriptor ( <b>Qw.d</b> ). With -qnoqcount, all <b>Q</b> edit descriptors are interpreted as the extended-precision <b>Q</b> edit descriptor. <b>Default: -qnoqcount</b>	161
-qrealize= <i>bytes</i>	REALSIZE( <i>bytes</i> )	Sets the default size of <b>REAL</b> , <b>DOUBLE PRECISION</b> , <b>COMPLEX</b> , and <b>DOUBLE COMPLEX</b> values. <b>Default: -qrealize=4</b>	162
-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 <b>STATIC</b> , while -qnosave sets it to <b>AUTOMATIC</b> . <b>Default: -qnosave</b>  Specify -qsave for <b>xl f</b> , <b>xl f_r</b> , <b>f77</b> , or <b>fort77</b> to duplicate the behaviour of FORTRAN77 commands.	167
-u -qundef -qnoundef	UNDEF NOUNDEF	Specifies whether implicit typing of variable names is permitted. -u and -qundef have the same effect as the <b>IMPLICIT NONE</b> statement that appears in each scope that allows implicit statements. <b>Default: -qnoundef</b>	194
-qxflag=oldtab	XFLAG(OLDTAB)	Interprets a tab in columns 1 to 5 as a single character (for fixed source form programs). <b>Default:</b> Tab is interpreted as one or more characters.	183

Table 7. Options for Compatibility (continued)

Command-Line Option	@PROCESS Directive	Description	See Page
-qxlf77= <i>settings</i>	XLF77( <i>settings</i> )	Provides compatibility with XL Fortran for AIX® Versions 1 and 2 aspects of language semantics and I/O data format that have changed. Most of these changes are required by the Fortran 90 standard. <b>Default:</b> Default suboptions are blankpad, nogedit77, nointarg, nointxor, leadzero, nooldboz, nopersistent, and nosofteof for the xlf90, xlf90_r, xlf95, and xlf95_r commands and are the exact opposite for the xlf, xlf_r, and f77/fort77 commands.	184
-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. <b>Default:</b> The default suboptions are <b>signedzero</b> and <b>autodealloc</b> for the xlf95 and xlf95_r invocation commands. For all other invocation commands, the default suboptions are <b>nosignedzero</b> and <b>noautodealloc</b> .	186

## Option for New Language Extensions

The following option is intended for writing new programs. It changes the program in ways that would otherwise require source-code changes.

Table 8. Option for Fortran 90 Extensions

Command-Line Option	@PROCESS Directive	Description	See Page
-qinit=f90ptr	INIT(f90ptr)	Makes the initial association status of pointers disassociated. <b>Default:</b> The default association status of pointers is undefined.	124

## 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 “-qfltrap Option” on page 117 and “Duplicating the Floating-Point Results of Other Systems” on page 209.

Table 9. 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. <b>Default:</b> Default suboptions are <b>nocomplexgcc</b> , <b>noflntint</b> , <b>fold</b> , <b>maf</b> , <b>nonans</b> , <b>norm</b> , <b>norsqrt</b> , and <b>nostrictmaf</b> ; some of these settings are different with <b>-O3</b> optimization turned on.	115
-qieee={ Near   Minus   Plus   Zero} -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. <b>Default:</b> <b>-qieee=near</b>	123

## 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 10. Options That Control Linking

Command-Line Option	@PROCESS Directive	Description	See Page
-c		Produces an object file instead of an executable file. <b>Default:</b> Compile and link-edit, producing an executable file.	67
-Ldir*		Looks in the specified directory for libraries specified by the <b>-l</b> option. <b>Default:</b> <b>/usr/lib</b>	74
-lkey*		Searches the specified library file, where <i>key</i> selects the file <b>libkey.dylib</b> or <b>libkey.a</b> . <b>Default:</b> Libraries listed in <b>xf.cfg</b> .	75
-qcommon -qnocommon	COMMON NOCOMMON	Controls whether global variables are allocated in the common or data section of the object file. <b>Default:</b> <b>-qcommon</b> allocates uninitialized global variables in the common section.	98

Table 10. Options That Control Linking (continued)

Command-Line Option	@PROCESS Directive	Description	See Page
-qp -qpic -qnopic		Generates Position Independent Code (PIC) that can be used in shared libraries. <b>Default:</b> -qp (At -O4, -O5, and -qipa, -qnopic is automatically turned on when you are not creating a shared library.)	157

## Options That Control the Compiler Internal Operation

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 11. Options That Control the Compiler Internal Operation

Command-Line Option	@PROCESS Directive	Description	See Page
-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. <b>Default:</b> Paths for these components are defined in the configuration file, the \$PATH environment variable, or both.	65
-Fconfig_file -Fconfig_file: stanza -F:stanza		Specifies an alternative configuration file, the stanza to use within the configuration file, or both. <b>Default:</b> The configuration file is /etc/opt/ibmcmp/xf/8.1/xf.cfg, and the stanza depends on the name of the command that executes the compiler.	70
-NSbytes -qSPILLsize= bytes	SPILLsize (bytes)	Specifies the size of internal program storage areas. <b>Default:</b> -NS512	76
-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. <b>Default:</b> -qmaxmem=2048; At -O3, -O4, and -O5, -qmaxmem=-1.	143
-tcomponents		Applies the prefix specified by the -B option to the designated components. <i>components</i> can be one or more of F, c, I, a, h, b, z, or l, with no separators, corresponding to the C preprocessor, the compiler, the interprocedural analysis (IPA) tool, the assembler, the loop optimizer, the code generator, the binder, and the linker, respectively. <b>Default:</b> -B prefix, if any, applies to all components.	192

Table 11. Options That Control the Compiler Internal Operation (continued)

Command-Line Option	@PROCESS Directive	Description	See Page
-Wcomponent,options		<p>Passes the listed options to a component that is executed during compilation. <i>component</i> is F, c, I, a, z, or l, corresponding to the C preprocessor, the compiler, the interprocedural analysis (IPA) tool, the assembler, the binder, and the linker, respectively.</p> <p><b>Default:</b> The options passed to these programs are as follows:</p> <ul style="list-style-type: none"> <li>• Those listed in the configuration file</li> <li>• Any unrecognized options on the command line (passed to the linker)</li> </ul>	197

## 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 12. Options That Are Obsolete or Not Recommended

Command-Line Option	@PROCESS Directive	Description	See Page
-qcharlen= length	CHARLEN (length)	<p>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).</p>	95
-qrecur -qnorecur	RECUR NORECUR	<p>Not recommended. Specifies whether external subprograms may be called recursively.</p> <p>For new programs, use the <b>RECURSIVE</b> keyword, which provides a standard-conforming way of using recursive procedures. If you specify the <b>-qrecur</b> option, the compiler must assume that any procedure could be recursive. Code generation for recursive procedures may be less efficient. Using the <b>RECURSIVE</b> keyword allows you to specify exactly which procedures are recursive.</p>	164

---

## 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 195 and “`-V` Option” on page 196 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 **-t***mnemonic* 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 **/Users/jim** and then tries it out with commands similar to:

```
/Users/jim/opt/ibmcmp/xlf/8.1/bin/xlf95 \  
-tFcbhIz -B/Users/jim/opt/ibmcmp/xlf/8.1/exe/ test_suite.f
```

Once the upgrade meets the acceptance criteria, the system administrator installs it over the old level in **/opt/ibmcmp/xlf/8.1**.

### Related Information

See “**-t** Option” on page 192, “**-F** Option” on page 70, and “Customizing the Configuration File” on page 15.

## -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.

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 122 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 191.

“-qsigtrap Option” on page 168 and “Installing an Exception Handler” on page 210 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 79.

## **-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 31.

## -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/8.1/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/8.1/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 /Users/fred/xlf.cfg.
xlf95 -F/Users/fred/xlf.cfg t.f

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

### Related Information

“Customizing the Configuration File” on page 15 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 | NODBG`

Generates debug information for use by a symbolic debugger.

### **Related Information**

See “Debugging a Fortran 90 or Fortran 95 Program” on page 265 and “Symbolic Debugger Support” on page 11.

`-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/8.1/include** where include and **.mod** files shipped with the compiler are located.

### Related Information

The “-qmoddir Option” on page 147 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 119 and *Free Source Form* in the *XL Fortran Advanced Edition for Mac OS X 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/8.1/lib**, you can specify one or more **-L** options that point to the locations of the other 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 59 and “Linking XL Fortran Programs” on page 32.

## **-l Option**

### **Syntax**

*-lkey*

Searches the specified library file, where *key* selects the library **libkey.dylib** 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 59 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 combination of 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=ppcv** 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.

### 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.

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 172 shows how to turn off the effects of -O3 that might change the semantics of a program.

“-qipa Option” on page 130, “-qhot Option” on page 122, and “-qpdf Option” on page 153 turn on additional optimizations that may improve performance for some programs.

“Optimizing XL Fortran Programs” on page 217 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.

### **Defaults**

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

### **Rules**

Except when you specify the `-c` 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"
```

## -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

```
.  
.  
.
```

### Related Information

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

## -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.

### 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**, and **xlF95\_r** 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 **POINTERS** may be present (**-qalias=aryovrlp:pteovrlp:std:intptr**).

## -qalign Option

### Syntax

```
-qalign={ [no]4k | struct={natural | packed | port} }  
ALIGN( { [NO]4K | STRUCT( { (natural) | (packed) | (port) } ) }
```

Specifies the alignment of data objects in storage, which avoids performance problems with misaligned data. Both the **[no]4k** and **struct** options can be specified and are not mutually exclusive. The default setting is **-qalign=no4k:struct=natural**. The **[no]4K** option is useful primarily in combination with logical volume I/O and disk striping.

### Arguments

#### **[no]4K**

Specifies whether to align large data objects on page (4 KB) boundaries, for improved performance with data-stripped 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.

**struct** 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 89. 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.

### Arguments

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.
- g5** Generates instructions specific to G5 processors. This is currently equivalent to specifying **-qarch=ppc970**.
- ppc970** Generates instructions specific to PowerPC 970 processors.
- ppcv** Generates instructions for generic PowerPC chips with AltiVec vector processors. This is the default.

**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=ppc970** to indicate that your program will be executed on Power PC 970 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 (**ppcv**). 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. The **-qarch** option may result in a program that cannot be run on machines with processors other than those supported by the option.
- 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 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=ppc970** when targeting a Power PC 970 machine will benefit those programs that are floating-point intensive or have integer multiplies.

### **Related Information**

See "Compiling for PowerPC Systems" on page 30, "-qtune Option" on page 179, and "-qcache Option" on page 92.

## -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 223 for background information and instructions for using these assertions. See also the description of the **ASSERT** directive in the *XL Fortran Advanced Edition for Mac OS X 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 51 and “Attribute and Cross-Reference Section” on page 270.

## -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:

<b><u>none</u></b>	Does not promote or pad any objects that share storage. This setting is the default.
<b>dbl4</b>	Promotes floating-point objects that are single-precision (4 bytes in size) or that are composed of such objects (for example, <b>COMPLEX</b> or array objects): <ul style="list-style-type: none"><li>• <b>REAL(4)</b> is promoted to <b>REAL(8)</b>.</li><li>• <b>COMPLEX(4)</b> is promoted to <b>COMPLEX(8)</b>.</li></ul> This suboption requires the <b>libxlfpm4.a</b> library during linking.
<b>dbl8</b>	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"><li>• <b>REAL(8)</b> is promoted to <b>REAL(16)</b>.</li><li>• <b>COMPLEX(8)</b> is promoted to <b>COMPLEX(16)</b>.</li></ul> This suboption requires the <b>libxlfpm8.a</b> library during linking.
<b>dbl</b>	Combines the promotions that <b>dbl4</b> and <b>dbl8</b> perform. This suboption requires the <b>libxlfpm4.a</b> and <b>libxlfpm8.a</b> libraries during linking.
<b>dblpad4</b>	Performs the same promotions as <b>dbl4</b> and pads objects of other types (except <b>CHARACTER</b> ) if they could possibly share storage with promoted objects. This suboption requires the <b>libxlfpm4.a</b> and <b>libxlfpad.a</b> libraries during linking.
<b>dblpad8</b>	Performs the same promotions as <b>dbl8</b> and pads objects of other types (except <b>CHARACTER</b> ) if they could possibly share storage with promoted objects. This suboption requires the <b>libxlfpm8.a</b> and <b>libxlfpad.a</b> libraries during linking.
<b>dblpad</b>	Combines the promotions done by <b>dbl4</b> and <b>dbl8</b> and pads objects of other types (except <b>CHARACTER</b> ) 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 pointer is acquired through the system routine **malloc**, the size specified to **malloc** should take into account the extra space needed to represent the pointer 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 284.

“**-qrealsize** Option” on page 162 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 **-qdpcc** option.

## -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 -03 -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 -03 -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 -03 -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 179, “-qarch Option” on page 86, and “-qhot Option” on page 122.

## **-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 **-qnocclines**.

### **Related Information**

See *Conditional Compilation* in the *Language Elements* section of the *XL Fortran Advanced Edition for Mac OS X Language Reference*.

## **-qcharlen Option**

### **Syntax**

`-qcharlen=length`  
`CHARLEN(length)`

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).

## **-qcheck Option**

### **Syntax**

`-qcheck` | `-qnocheck`  
`CHECK` | `NOCHECK`

`-qcheck` is the long form of the “-C Option” on page 66.

## -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 Advanced Edition for Mac OS X Language Reference*.

## **-qcommon Option**

### **Syntax**

**-qcommon** | **-qnocommon**  
**COMMON** | **NOCOMMON**

Controls whether global variables are allocated in the common or data section of object file.

By default, all uninitialized global variables are allocated in the common section. When **-qnocommon** is set, global variables are allocated in the data section of the object file. Setting **-qnocommon** prevents global variables from being simultaneously defined in different object files. Common symbols cannot be used in shared libraries.

## **-qcompact Option**

### **Syntax**

-qcompact		-qnocompact
COMPACT		<u>NOCOMPACT</u>

Reduces optimizations that increase code size.

By default, some techniques the optimizer uses to improve performance may also make the program larger. For systems with limited storage, you can use **-qcompact** to reduce the expansion that takes place.

### **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 Advanced Edition for Mac OS X Language Reference* and “Passing Arguments By Reference or By Value” on page 248.

## **-qdbg Option**

### **Syntax**

-qdbg | -qnodbg  
DBG | **NODBG**

**-qdbg** is the long form of the “-g Option” on page 71.

## -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 directive is a line that is not a Fortran statement but is recognized and acted on by the compiler. The compiler always recognizes some directives, such as @PROCESS. 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\***.

### 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

```
@PROCESS FREE
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 that has not
! been converted yet.
    INCLUDE 'set_c_and_d.inc'
!IBM* SOURCEFORM(FREE)
E = 5 ! Back to free source form.
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 converting the include file to use free source form, you can compile with the **-qnodirective** option, and the **SOURCEFORM(FIXED)** directive is ignored.

## Related Information

See the section on the **SOURCEFORM** directive in the *XL Fortran Advanced Edition for Mac OS X 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 Advanced Edition for Mac OS X Language Reference* for the suitable associated trigger constant.

## **-qdlines Option**

### **Syntax**

`-qdlines` | `-qnodlines`  
`DLINES` | `NODLINES`

`-qdlines` is the long form of the “-D Option” on page 68.

## -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 90 and “-qrealsize Option” on page 162 are more general-purpose options that can also do what **-qdp** does. **-qdp** has no effect if you specify either of these options.

## -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\bcd\efg'
      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\bcd\efg
```

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 19 on page 247.

## -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 110 and “-qsigtrap Option” on page 168.

## **-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 f`, `xl f_r`, `f77`, and `fort77` commands. `-qfree=f90` is the default for the `xl f90`, `xl f90_r`, `xl f95`, and `xl f95_r` commands.

### **Related Information**

See “-qfree Option” on page 119.

For the precise specifications of this source form, see *Fixed Source Form* in the *XL Fortran Advanced Edition for Mac OS X Language Reference*.

## -qflag Option

### Syntax

`-qflag=listing_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 136 and “Understanding XL Fortran Error Messages” on page 259.

## -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 201 and the IEEE standard before attempting to change any `-qfloat` settings.

### Defaults

The default setting uses the suboptions `nocomplegcc`, `nofltint`, `fold`, `maf`, `nonans`, `norm`, `norsqrt`, 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 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` | `nocomplegcc`

Use Mac OS X 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`.

**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.

**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 | norm**

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**.

## -qfltrap Option

### Syntax

`-qfltrap[=suboptions] | -qnofltrap`  
`FLTRAP[(suboptions)] | 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.

### Arguments

<b>O</b> verflow	Detect and trap on floating-point overflow if exception-checking is enabled.
<b>UN</b> Derflow	Detect and trap on floating-point underflow if exception-checking is enabled.
<b>ZER</b> Odivide	Detect and trap on floating-point division by zero if exception-checking is enabled.
<b>IN</b> Valid	Detect and trap on floating-point invalid operations if exception-checking is enabled.
<b>IN</b> EXact	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.
<b>EN</b> able	Turn on checking for the specified exceptions in the main program so that the exceptions generate <b>SIGFPE</b> signals. You must specify this suboption if you want to turn on exception trapping without modifying your source code.
<b>IM</b> Precise	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.

### Defaults

The **-qfltrap** option without suboptions is equivalent to **-qfltrap=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 **-qfltrap** more than once, both with and without suboptions, the **-qfltrap** without suboptions is ignored.

## Examples

When you compile this program:

```
REAL X, Y, Z
DATA X /5.0/, Y /0.0/
Z = X / Y
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 168.

See “Detecting and Trapping Floating-Point Exceptions” on page 209 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 **xlfc90**, **xlfc90\_r**, **xlfc95**, and **xlfc95\_r** commands.

### Related Information

See “-qfixed Option” on page 113.

**-k** is equivalent to **-qfree=f90**.

Fortran 90 free source form is explained in *Free Source Form* in the *XL Fortran Advanced Edition for Mac OS X 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 Advanced Edition for Mac OS X 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 71.

## -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** A severity level that can never be generated by any defined error condition. Specifying it prevents the compiler from halting, 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`

Determines whether to perform high-order transformations on loops and array language during optimization and whether to pad array dimensions and data objects to avoid cache misses.

Because of the implementation of the cache architecture, array dimensions that are powers of two can lead to decreased cache utilization. The optional **arraypad** suboption 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**.

If you do not specify at least level 2 of **-O** for **-qhot**, the compiler assumes **-O2**.

### Arguments

#### **arraypad**

The compiler will pad any arrays where it infers there may be a benefit and will pad by whatever amount it chooses. Not all arrays will necessarily be padded, and different arrays may be padded by different amounts.

#### **arraypad=*n***

The compiler will pad every array in the code. The pad amount must be a positive integer value. Each array will be padded by an integral number of elements.

Because *n* is an integral value, we recommend that pad values be multiples of the largest array element size, typically 4, 8, or 16.

For both the **arraypad** and **arraypad=*n*** options, no checking for reshaping or equivalences is performed. If padding takes place, your compiled program may produce unpredictable results.

### Restrictions

The **-C** option turns off the transformations.

### Related Information

“Optimizing Loops and Array Language” on page 223 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:

<b>Near</b>	Round to nearest representable number.
<b>Minus</b>	Round toward minus infinity.
<b>Plus</b>	Round toward plus infinity.
<b>Zero</b>	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 206, “-O Option” on page 77, and “-qfloat Option” on page 115.

## **-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.

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 Advanced Edition for Mac OS X 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 **7FBFFFFFF**, 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 “-qfltrap Option” on page 117 and the section on the **AUTOMATIC** directive in the *XL Fortran Advanced Edition for Mac OS X 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 size<sup>1</sup> 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.

### 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 162 and *Type Parameters and Specifiers* in the *XL Fortran Advanced Edition for Mac OS X 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**.) 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 phase 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 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.

If compiling and linking are performed in the same step and you do not specify 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 153 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**.

**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.

## 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.
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 .\$ would show all lines that had at least one character. Specify \.\$ to escape the period (.).
[string]	Matches any of the characters specified in string. For example, t[a-g123]st matches tast and test, but not t-st or tAst.

Table 13. Regular expression syntax (continued)

Expression	Description
[^string]	Does not match any of the characters specified in string. For example, t[^a-zA-Z]st matches t1st, t-st, and t,st but not test or tYst.
string*	Matches zero or more occurrences of the pattern specified by string. For example, te*st will match tst, test, and teeeest.
string+	Matches one or more occurrences of the pattern specified by string. For example, t(es)+t matches test, tesest, but not tt.
string?	Matches zero or more occurrences of the pattern specified by string. For example, te?st matches either tst or test.
string{m,n}	Matches between m and n occurrence(s) of the pattern specified by string. For example, a{2} matches aa, b{1,4} matches b, bb, bbb, and bbbb.
string1   string2	Matches the pattern specified by either string1 or string2. For example, s   o matches both characters s and o.

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 77 and “-p Option” on page 80.

## **-qkeepparm Option**

### **Syntax**

`-qkeepparm`

### **Background Information**

A procedure usually stores its incoming parameters on the stack at the entry point. When you compile code with `-O`, 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**

<b>77std</b>	Accepts the language that the ANSI FORTRAN 77 standard specifies and reports anything else as an error.
<b>90std</b>	Accepts the language that the ISO Fortran 90 standard specifies and reports anything else as an error.
<b>90pure</b>	The same as <b>90std</b> except that it also reports errors for any obsolescent Fortran 90 features used.
<b>95std</b>	Accepts the language that the ISO Fortran 95 standard specifies and reports anything else as an error.
<b>95pure</b>	The same as <b>95std</b> except that it also reports errors for any obsolescent Fortran 95 features used.
<u><b>extended</b></u>	Accepts the full Fortran 95 language standard plus 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:

<b>-qlanglvl Suboption Specified</b>	<b>Result</b>	<b>Reason</b>
<b>95pure</b>	Flags <b>PAUSE</b> statement Flags computed <b>GOTO</b> statement Flags <b>GAMMA</b> intrinsic	Deleted feature in Fortran 95 Obsolescent feature in Fortran 95 Extension to Fortran 95
<b>95std</b>	Flags <b>PAUSE</b> statement Flags <b>GAMMA</b> intrinsic	Deleted feature in Fortran 95 Extension to Fortran 95
<b>extended</b>	No errors flagged	

## Related Information

See “`-qflag` Option” on page 114, “`-qhalt` Option” on page 121, and “`-qsaa` Option” on page 166.

The `langlvl` run-time option, which is described in “Setting Run-Time Options” on page 34, helps to locate run-time extensions that cannot be checked for at compile time.

## **-qlibansi Option**

### **Related Information**

See “-qipa Option” on page 130.

## **-qlibposix Option**

### **Related Information**

See “-qipa Option” on page 130.

## **-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 51 and “Object Section” on page 271.

## -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 51 and “Options Section” on page 267.

## **-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 77 and “Optimizing XL Fortran Programs” on page 217.

## -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.

## **-qmixed Option**

### **Syntax**

`-qmixed` | `-qnomixed`  
MIXED | NOMIXED

This is the long form of the “-U Option” on page 193.

## **-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.

### **Related Information**

See “XL Fortran Output Files” on page 26.

Modules are a Fortran 90/95 feature and are explained in the *XL Fortran Advanced Edition for Mac OS X Language Reference*.

To read the **.mod** files from this directory when compiling files that reference the modules, use the “-I Option” on page 72.

## **-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 51.

## -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 246.

## -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 51 and “Object Section” on page 271.

“The Compiler Phases” on page 283 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 64.

## **-qoptimize Option**

### **Syntax**

`-qOPTimize[=level] | -qNOOPTimize`  
`OPTimize[(level)] | NOOPTimize`

This is the long form of the “-O Option” on page 77.

## -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 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.

## 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 25, “XL Fortran Output Files” on page 26, and “Optimizing Conditional Branching” on page 226.

The following commands, in the directory `/opt/ibmcomp/xlf/8.1/bin`, are available for managing the **PDFDIR** directory:

`resetpdf [pathname]` Sets to zeros all profiling information (but does not remove the data files) from the *pathname* directory, or from the **PDFDIR** directory if *pathname* is not specified, or from the current directory if **PDFDIR** is not set.

When you make changes to the application and recompile some files, the profiling information for those files is automatically reset because the changes may alter the program flow. Run **resetpdf** to reset the profiling information for the entire application after you make significant changes that may change execution counts for parts of the program that were not recompiled.

`cleanpdf [pathname]` Removes all profiling information from the *pathname* directory, or from the **PDFDIR** directory if *pathname* is not specified, or from the current directory if **PDFDIR** is not set.

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**.

## **-qphsinfo Option**

### **Syntax**

`-qphsinfo` | `-qnophsinfo`  
`PHSINFO` | `NOPHSINFO`

The `-qphsinfo` compiler option displays timing information on the terminal for each compiler phase.

### **Related Information**

“The Compiler Phases” on page 283.

## **-qpic Option**

### **Syntax**

qpic|qnopic

The **-qpic** compiler option generates Position Independent Code (PIC) that can be used in shared libraries.

## -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.

**sce** | **nosce** Specifying this option allows the compiler to perform short circuit evaluation in selected logical expressions.

**typestmt** | **notypestmt**

The **TYPE** statement, which behaves in a manner similar to the **PRINT** statement, is supported whenever this option is specified.

**typlssarg** | **notyplssarg**

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 Advanced Edition for Mac OS X 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 235 and the section on the **OPEN** statement in the *XL Fortran Advanced Edition for Mac OS X 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 Advanced Edition for Mac OS X Language Reference*. To selectively control prefetch directives using trigger constants, see the “-qdirective Option” on page 105.

## **-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 Advanced Edition for Mac OS X 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 sizes<sup>2</sup> 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 128 is a similar option that affects integer and logical objects. “-qautodbl Option” on page 90 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 Advanced Edition for Mac OS X 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 **xlf**, **xlf\_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[={hotlist}...]  
-qnoreport  
REPORT[({HOTLIST}...)] NOREPORT
```

Determines whether to produce transformation reports showing how loops are optimized.

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 **hotlist** suboption to generate a report showing how loops are transformed.

### Arguments

**hotlist** 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.

### Background Information

The transformation listing is part of the compiler listing file.

### Restrictions

Loop transformation is 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 **-qhot** option to generate a loop transformation listing.

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 the performance of loops:

```
xlf95 -O3 -qhot -qreport=hotlist needs_tuning.f
```

### Related Information

See “Options That Control Listings and Messages” on page 51 and “Transformation Report Section” on page 269.

## **-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 136 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/8.1/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 164.

See *Storage Classes for Variables* in the *XL Fortran Advanced Edition for Mac OS X Language Reference* for information on how this option affects the storage class of variables.

## **-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 41. “Detecting and Trapping Floating-Point Exceptions” on page 209 describes a number of methods for dealing with exceptions that result from floating-point computations. “Installing an Exception Handler” on page 210 lists the exception handlers that XL Fortran supplies.

## **-qsmallstack Option**

### **Syntax**

`-qsmallstack` | `-qnosmallstack`

Specifies that the compiler will minimize stack usage where possible.

## -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 51 and "Source Section" on page 268.

## **-qspillsize Option**

### **Syntax**

`-qspillsize=bytes`  
`SPILLSIZE(bytes)`

`-qspillsize` is the long form of `-NS`. See “`-N Option`” on page 76.

## -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 77, “-qhot Option” on page 122, and “-qfloat Option” on page 115.

## -qstrictieemod Option

### Syntax

`-qstrictieemod` | `-qnostrictieemod`  
`STRICTIEEEMOD` | `NOSTRICTIEEEMOD`

Specifies whether the compiler will adhere to the draft Fortran 2000 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
i = 1_1                     ! Variable i can hold a maximum
                             ! 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 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.

### **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.

### **Arguments**

<b>auto</b>	Automatically detects the specific processor type of the compiling machine. It assumes that the execution environment will be the same as the compilation environment.
<b>g5</b>	Generates object code optimized for G5 processors. This is currently equivalent to specifying <b>-qtune=ppc970</b> .
<b>ppc970</b>	Generates object code optimized for PowerPC 970 processors. This is the default.

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.

### **Related Information**

See “**-qarch Option**” on page 86, “**-qcache Option**” on page 92, and “**Compiling for PowerPC Systems**” on page 30.

## **-qundef Option**

### **Syntax**

`-qundef` | `-qnoundef`  
`UNDEF` | `NOUNDEF`

`-qundef` is the long form of the “-u Option” on page 194.

## -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 Advanced Edition for Mac OS X Language Reference*.

- **STREAM\_UNROLL**
- **UNROLL**
- **UNROLL\_AND\_FUSE**

See "Optimizing Loops and Array Language" on page 223.

## -qunwind Option

### Syntax

<code>-qunwind</code>		<code>-qnounwind</code>
<code><u>UNWIND</u></code>		<code>NOUNWIND</code>

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.

## **-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 XL Fortran for AIX Versions 1 and 2 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 `xlf95`, `xlf95_r`, `xlf90`, and `xlf90_r` 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 `xlf90`, `xlf90_r`, `xlf95`, or `xlf95_r` 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**

Does not produce 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.

## -qxf90 Option

### Syntax

`-qxf90={settings}`  
`XLf90({settings})`

Provides compatibility with XL Fortran for AIX Version 5 and the Fortran 90 standard for certain aspects of the Fortran language.

### Defaults

The default suboptions for `-qxf90` depend on the invocation command that you specify. For the `xf95` or `xf95_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 `-qxf90=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 section on **SIGN** in the *Intrinsic Procedures* section and the *Arrays Concepts* section of the *XL Fortran Advanced Edition for Mac OS X 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.

### 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 68 and *Conditional Compilation in the Language Elements* section of the *XL Fortran Advanced Edition for Mac OS X 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 51 and “Attribute and Cross-Reference Section” on page 270.

## -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**, and **xlF95\_r** 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.

## -t Option

### Syntax

*-tcomponents*

Applies the prefix specified by the **-B** option to the designated components. *components* can be one or more of **F**, **c**, **h**, **I**, **a**, **b**, **l**, or **z** with no separators, corresponding to the C preprocessor, the compiler, the array language optimizer, the interprocedural analysis (IPA) tool/loop optimizer, the assembler, the code generator, the linker, and the binder, respectively.

### Rules

If **-t** is not specified, any **-B** prefix is applied to all components.

Component	-t Mnemonic	Standard Program Name
C preprocessor	F	cpp
compiler front end	c	xlfcntry
array language optimizer	h	xlshot
IPA/loop optimizer	I	ipa
assembler	a	as
code generator	b	xlfcode
linker	l	ld
binder	z	bolt

### Related Information

See “-B Option” on page 65 (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 146.

## **-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 Advanced Edition for Mac OS X Language Reference*.

This is the short form of **-qundef**. See “-qundef Option” on page 180.

## **-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 63 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 **F**, **c**, **I**, **a**, **z**, or **l**, corresponding to the C preprocessor, the compiler, the interprocedural analysis (IPA) tool, the assembler, the binder, and the linker, respectively.

In the string following the **-W** option, use a comma as the separator, and do not include any spaces.

### 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.

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 29.

## **-w Option**

### **Syntax**

`-w`

A synonym for the “-qflag Option” on page 114. 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 77 and “-qfloat Option” on page 115.

`-y` is the short form of “-qieee Option” on page 123.



---

## 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 59, especially “-qfloat Option” on page 115. 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 Advanced Edition for Mac OS X 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 2000 IEEE Module and arithmetic support, see the *XL Fortran Advanced Edition for Mac OS X 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 Advanced Edition for Mac OS X 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 205 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 209 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 206 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 209) 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 209.

### 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/8.1/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 **xlfp\_util** module. These procedures return and set the current floating-point rounding mode, respectively.

For example:

```

program fptest
  use 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 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/8.1/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/8.1/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 206 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 162) 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** and **nohssngl**. 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 204.

## Installing an Exception Handler

When a program that uses the XL Fortran or Mac OS X 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/8.1/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:

<b>xl_ieee</b>	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.
<b>xl_trce</b>	Produces a traceback and stops the program.
<b>xl_trcedump</b>	Produces a traceback and a core file and stops the program.
<b>xl_sigdump</b>	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.
<b>xl_trbk</b>	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 <b>-qsigtrap</b> 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/8.1/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 214 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 Advanced Edition for Mac OS X Language Reference*.

## Controlling the Floating-Point Status and Control Register

Before the `-qflttrap` 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 `-qflttrap` option:

```
xlf95 -qflttrap compute_pi.f # Check all exceptions, but do not trap.
xlf95 -qflttrap=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/ibmcmp/xlf/8.1/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 206.

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 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 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 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 modules `xlf_fp_util`:

File names	File type	Locations
<code>xlf_fp_util.mod</code>	module symbol file	<code>/opt/ibmcmp/xlf/8.1/include</code>

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 Advanced Edition for Mac OS X 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 Advanced Edition for Mac OS X 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	<code>fpstat(fpx)</code>	Floating-point exception summary
n/a	<code>fpstat(fpfex)</code>	Floating-point enabled exception summary
<code>fpstat(fpve)</code>	<code>fpstat(fpvx)</code>	Floating-point invalid operation exception summary
<code>fpstat(fpoe)</code>	<code>fpstat(fpox)</code>	Floating-point overflow exception
<code>fpstat(fpue)</code>	<code>fpstat(fpux)</code>	Floating-point underflow exception
<code>fpstat(fpze)</code>	<code>fpstat(fpzx)</code>	Zero-divide exception
<code>fpstat(fpxe)</code>	<code>fpstat(fpxx)</code>	Inexact exception
<code>fpstat(fpve)</code>	<code>fpstat(fpvxsnan)</code>	Floating-point invalid operation exception (signaling NaN)
<code>fpstat(fpve)</code>	<code>fpstat(fpvxisi)</code>	Floating-point invalid operation exception (INF-INF)
<code>fpstat(fpve)</code>	<code>fpstat(fpvxidi)</code>	Floating-point invalid operation exception (INF/INF)
<code>fpstat(fpve)</code>	<code>fpstat(fpvxzdz)</code>	Floating-point invalid operation exception (0/0)
<code>fpstat(fpve)</code>	<code>fpstat(fpvximz)</code>	Floating-point invalid operation exception (INF*0)
<code>fpstat(fpve)</code>	<code>fpstat(fpvxvc)</code>	Floating-point invalid operation exception (invalid compare)
n/a	<code>fpstat(fpvxsoft)</code>	Floating-point invalid operation exception (software request)

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(fpvxsqrt)</code>	Floating-point invalid operation exception (invalid square root)
n/a	<code>fpstat(fpvx cvi)</code>	Floating-point invalid operation exception (invalid integer convert)

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(fpfx)`, 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/8.1/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.

### `fhandler.F`

Shows a sample Fortran signal handler and demonstrates the `xl_sigdump` procedure.

`xl__trbk_test.f`

Shows how to use the `xl__trbk` 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:

```
xlf90 -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.



---

## 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 compilers are capable of widely applicable and sophisticated program analysis and transformation.

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.<sup>3</sup> 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 **-qfloat=nomaf** or **-qfloat=rrm**.

---

## 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 no optimizations (**-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
<b>-qnoopt/-O0</b>	Fast compilation, debugable code, conserved program semantics.
<b>-O2</b>	Comprehensive low-level optimization; partial debugging support. (This is the same as specifying <b>-O</b> .)
<b>-O3</b>	More extensive optimization; some precision trade-offs.
<b>-O4 and -O5</b>	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.

## 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 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
<b>-qarch</b>	Selects a family of processor architectures, or a specific architecture, for which instruction code should be generated.
<b>-qtune</b>	Biases optimization toward execution on a given processor, without implying anything about the instruction set architecture to use as a target.
<b>-qcache</b>	Defines a specific cache or memory geometry. The defaults are set through <b>-qtune</b> .

Selecting a predefined optimization level sets default values for these individual options.

**Related Information:** See “**-qarch Option**” on page 86, “**-qtune Option**” on page 179, and “**-qcache Option**” on page 92.

## 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=ppcv**.
- 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.

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.

If you decide to use **-qcache**, use **-qhot** 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 **-qflttrap** enables software-based checking. On the Power PC 970 processor, hardware trapping is recommended.

Options for handling floating-point calculations

Option	Description
<b>-qfloat</b>	Provides precise control over the handling of floating-point calculations.
<b>-qflttrap</b>	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 209 and “Minimizing the Performance Impact of Floating-Point Exception Trapping” on page 215.

---

## 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.

## 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 **-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.

You can use the **-qhot** option 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 181.

## 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, **ppc970** 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 PowerPC cache architecture, 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 122.

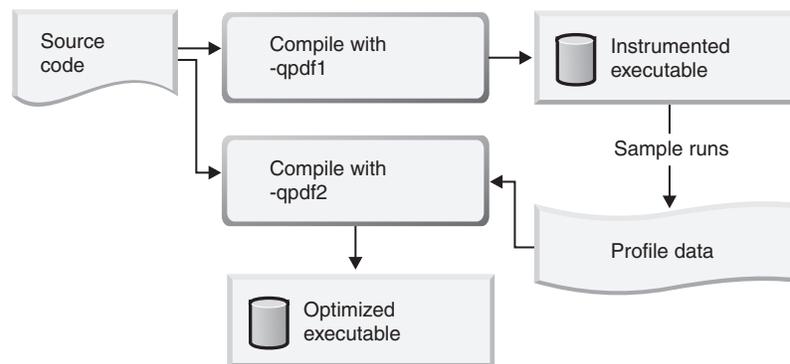
---

## 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.

## 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 153.

---

## 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

<b>Suboption</b>	<b>Behavior</b>
<b>level=0</b>	Program partitioning and simple interprocedural optimization, which consists of: <ul style="list-style-type: none"><li>• Automatic recognition of standard libraries.</li><li>• Localization of statically bound variables and procedures.</li><li>• 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.)</li><li>• Expansion of scope for some optimizations, notably register allocation.</li></ul>
<b>level=1</b>	Inlining and global data mapping. Specifically, <ul style="list-style-type: none"><li>• Procedure inlining.</li><li>• Partitioning and layout of static data according to reference affinity. (Data that is frequently referenced together will be located closer together in memory.)</li></ul> <p>This is the default level when <b>-qipa</b> is specified.</p>
<b>level=2</b>	Global alias analysis, specialization, interprocedural data flow. <ul style="list-style-type: none"><li>• 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.</li><li>• 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.</li><li>• Interprocedural constant propagation, dead code elimination, pointer analysis.</li><li>• Procedure specialization (cloning).</li></ul>
<b>inline=inline-options</b>	Provides precise user control of inlining.
<i>fine_tuning</i>	Other values for <b>-qipa=</b> 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 (**xlf**, **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`.
- If your application contains a mixture of Fortran and C or C++ code compiled with IBM 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 source files, include the `-qipa` option also.

```
# 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 “`-qipa` Option” on page 130.

## 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 files. 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.

---

## 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
<b>-qstrict, -qnostrict</b>	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 <b>-qstrict</b> with <b>-qnoopt</b> and <b>-O2</b> ; <b>-qnostrict</b> with <b>-O3</b> , <b>-O4</b> , and <b>-O5</b> .
<b>-qalias</b>	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
<b>-qcompact</b>	Chooses reduction of final code size over a reduction in execution time when a choice is necessary. Can be used to constrain <b>-O3</b> and higher optimizations.
<b>-qsmallstack</b>	Instructs the compiler to limit the use of stack storage in the program. Doing so may increase heap usage.
<b>-qunroll</b>	Independently controls loop unrolling. Is implicitly activated under <b>-O3</b> and higher optimizations.
<b>-qunwind</b>	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.
<b>-qnounwind</b>	Informs the compiler that the stack will not be unwound while any routine in this compilation is active. The <b>-qnounwind</b> 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
<b>-qlist</b>	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.
<b>-qreport</b>	Instructs the compiler to produce a report of the loop transformations it performed and how the program was parallelized. The option is enabled when <b>-qhot</b> is specified.
<b>-qinitauto</b>	Instructs the compiler to emit code that initializes all automatic variables to a given value.
<b>-qipa=list</b>	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 71, “-qinitauto Option” on page 125, and “Problem Determination and Debugging” on page 259.

---

## 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 ensure standards conformance and enhance 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** library 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. If you must use 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. Use the **REAL(16)**, or **COMPLEX(32)** data type only when extremely high precision is required.
- 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 Mac OS X file system.

**Related Information:** See “-qposition Option” on page 159 and “Mixed-Language Input and Output” on page 242.

---

## 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.

### **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 Mac OS X 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 C, “XL Fortran Internal Limits,” on page 291.)
- 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 1023 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. File names on Mac OS X are case-insensitive. The files **test.f** and **test.F** are considered to have the same name and cannot be in the same directory. The XL Fortran compiler, however, recognizes case-sensitive file names.

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 17. 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
<b>option not specified</b>	Start	Start	Error	Start	Start	Error	Start	Start
<b>appendold</b>	Start	Start	Error	Start	End	Error	Start	Start
<b>appendunknown</b>	Start	Start	Error	Start	Start	Error	End	Start
<b>appendold and appendunknown</b>	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.

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 34, 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**. 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:cverr=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\_** subroutine to write any buffered data to a file:

```

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 242.

---

## 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)

XLFRTSEOPTS="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 XLFRTSEOPTS 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 `XLFRTEOPTS` variable, depending on what other run-time options you have set. See “Setting Run-Time Options” on page 34 for other options that might be part of the `XLFRTEOPTS` 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 `XLFSCRATCH_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)
XLFRTEOPTS="scratch_vars=yes"      # Turn on scratch file naming.
XLFSCRATCH_1="/tmp/molecules.dat"  # Use this named file.
XLFSCRATCH_12="./data/scratch"     # Relative to current directory.
XLFSCRATCH_123="/home/user/data"   # Somewhere besides /tmp.
export XLFRTEOPTS XLFSCRATCH_1 XLFSCRATCH_12 XLFSCRATCH_123

```

**Notes:**

1. The `XLFSCRATCH_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 `XLFRTEOPTS` variable, depending on what other run-time options you have set. See “Setting Run-Time Options” on page 34 for other options that might be part of the `XLFRTEOPTS` value.
3. If the `scratch_vars` run-time option is set to `no` or is undefined or if the applicable `XLFSCRATCH_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.



---

## 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, `_MOD_`, and the name of the module procedure. For example, module procedure `MYPROC` in module `MYMOD` has the external name `__mymod_MOD_myproc`.

- 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 111.

---

## 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 Advanced Edition for Mac OS X Language Reference*. For more information on the **buffering** run-time option, see “Setting Run-Time Options” on page 34.

---

## 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 **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 248 and %VAL and %REF in the *XL Fortran Advanced Edition for Mac OS X 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.

### Passing Arguments Between Languages

Table 18. 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> 1. Requires a C compiler that supports 128-bit long double. 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 248.
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/8.1/samples` illustrate how to call from Fortran to C.

## 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                        main() {
.                                                }
end
```

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.

## 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 149.

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 19. 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	x, where x 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 20. 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.dylib**, 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 intermediate values.

#### Conventions for Passing by Value

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.

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 Mac OS X conventions when passing or returning complex numbers. `-qfloat=nocomplexgcc` is the default.

For `-qfloat=complexgcc`, the compiler passes `COMPLEX *8` values in 2 general-purpose registers (GPRs) and `COMPLEX *16` values in 4 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`, `COMPLEX *8` values are returned in GPR3-GPR4, and `COMPLEX *16` in GPR3-GPR6. 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 256.

## 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 *Inside Mac OS X: Mach-O Runtime Architecture* is the base reference on this topic. You should consult it 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 21. 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.
⋮	⋮	⋮
13	no	FP parameter 13, function return 13.
14-31	yes	

The following table lists general-purpose registers and their functions.

*Table 22. General-Purpose Register Usage across Calls*

Register	Preserved Across Calls	Use
0	no	
1	yes	Stack pointer.
2	no	
3	no	1st word of arg list; return value 1.
4	no	2nd word of arg list; return value 2.
⋮	⋮	⋮
10	no	8th word of arg list.
11	no	
12	no	

Table 22. General-Purpose Register Usage across Calls (continued)

Register	Preserved Across Calls	Use
13-31	yes	
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 23. 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 diagram shows the storage map of a typical stack frame.

In this diagram, 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, 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 16-byte aligned. The FPR save area and the parameter area (P1, P2, ..., Pn) are double-word aligned. Other areas require word alignment only.

**Run-time Stack  
Vector Information not Included**

Low Addresses			Stack grows at this end.
Callee's stack pointer	--> 0 4 8 12-16 20	Back chain Saved CR Saved LR Reserved	<--- LINK AREA (callee)
Space for P1-P8 is always reserved	24	P1 ... Pn	OUTPUT ARGUMENT AREA <---(Used by callee to construct argument list)
		Callee's stack area	<--- LOCAL STACK AREA
-8* <i>nfprs</i> -4* <i>ngprs</i> save	-->	Save area for caller's GPR max 19 words	(Possible word wasted for alignment.) Rfirst = R13 for full save R31
-8* <i>nfprs</i>	-->	Save area for caller's FPR max 18 dblwds	Ffirst = F14 for a full save F31
Caller's stack pointer	--> 0 4 8 12-16 20	Back chain Saved CR Saved LR Reserved	<--- LINK AREA (caller)
Space for P1-P8 is always reserved	24	P1 ... Pn	INPUT PARAMETER AREA <---(Callee's input parameters found here. Is also caller's arg area.)
High Addresses		Caller's stack area	

## The Link Area

The link area consists of six words at offset zero from the caller's stack pointer on entry to a procedure. The first word contains the caller's back chain (stack pointer). The second word is the location where the callee saves the Condition Register (CR) if it is needed. The third word is the location where the callee's prolog code saves the Link Register if it is needed. The fourth word is reserved for C **SETJMP** and **LONGJMP** processing, and the fifth word is reserved for future use. The last word (word 6) is reserved.

## The Input Parameter Area

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 words in size. If more than 8

words of parameters are expected, they are stored as register images that start at positive offset 56 from the incoming 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.

## The Register Save Area

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 above 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:

18\*8 for FPRs + 19\*4 for GPRs

Locations at a numerically lower address than the stack floor should not be accessed.

A callee needs only to save the nonvolatile registers that it actually uses. It always saves register 31 in the highest addressed word.

## 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

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 words long, regardless of the length or existence of any argument list. If more than 8 words are being passed, an extension list is constructed beginning at offset 56 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.

---

## 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.

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:

#### Conventions

- Each **INTEGER(8)** and **LOGICAL(8)** argument requires two words.
- Any other non-floating-point scalar argument of intrinsic type requires one word and appears in that word exactly as it would appear in a GPR. It is right-justified, 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 appear in successive words as they would anywhere in storage, satisfying all appropriate alignment requirements. Structures are aligned to a fullword and occupy  $(\text{sizeof}(\text{struct } X)+3)/4$  fullwords, with any padding at the end. A structure that is smaller than a word is left-justified within its word 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 257 for more information.)

## Argument Passing Rules (by Value)

From the following illustration, we state these rules:

- The parameter list is a conceptually contiguous piece of storage that contains a list of words. For efficiency, the first 8 words of the list are not actually stored in the space that is reserved for them but are passed in GPR3-GPR10. Further, the first 13 floating-point value parameters are passed in FPR1-FPR13. Those beyond the first 8 words of the parameter list are also in storage. Those within the first 8 words of the parameter list have GPRs reserved for them, but they are not used.
- 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.
- 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(l1), %val(l2), %val(l3), %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)

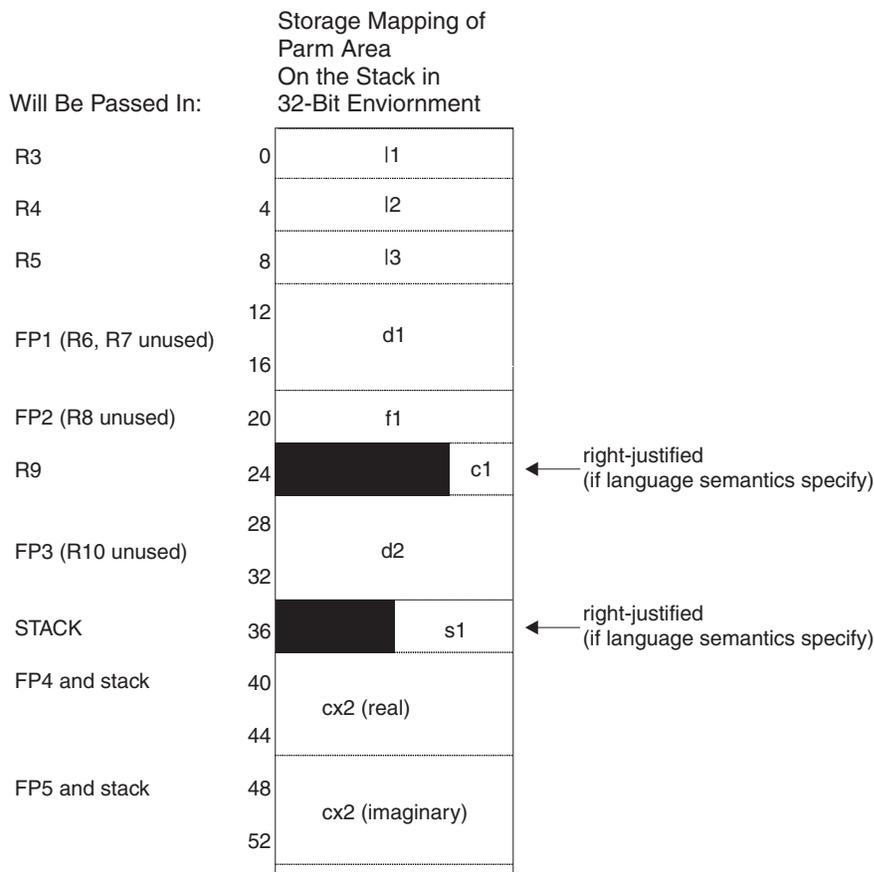


Figure 4. Storage Mapping of Parm Area On the Stack

## 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

The compiler handles calls to functions in the same compilation unit and calls to imported functions differently.

Calls to functions in the same compilation unit are turned into a branch to the address of the function:

---

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.

```
bl foo ; branch to function foo
```

Calls to imported functions are expanded into a branch to a function stub:

```
bl 0x60 ; symbol stub for foo
```

The function stub is located in the `picsymbol_stub` section of the object. The compiler generates a stub for each unique, imported function. The stub contains the instructions that load the address of the routine and branch to it (the routine).

If at link time the object with the imported function is linked statically, the linker will bypass the stub and rewrite the `bl` instruction so that the instruction branches directly to the routine. This means that only calls to dynamic library routines will go through a stub. All other calls will be direct calls in the final executable.

## Pointers to Functions

Function pointers are 4 bytes long and contain a 32-bit address. For pointers to local functions, the address contained is the address of the function in the text section. For imported functions, the address is that of the function's stub. Every unique, imported function will have a stub in the object. The function stub is in the non-lazy symbol pointer section.

## Function Values

Functions return their values according to type:

- **INTEGER** and **LOGICAL** of kind 1, 2, and 4 are returned (right justified) in R3.
- **INTEGER** and **LOGICAL** of kind 8 are returned in R3 and R4.
- **REAL** of kind 4 or 8 are returned in FP1. **REAL** are returned in FP1 and FP2.
- **COMPLEX** of kind 4 or 8 are returned in FP1 and FP2. **COMPLEX** of kind 16 are returned in FP1-FP4.
- When `-qfloat=complexgcc` is specified, **COMPLEX** of kind 8 is returned in R3-R4 and **COMPLEX** of kind 16 in R3-R6.
- 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

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 at offset 8 from the stack pointer if necessary.
2. If you use any of the CR bits 8-19 (CR2, CR3, CR4), save the CR at displacement 4 from the current stack pointer.
3. Save any nonvolatile FPRs that are used by this procedure in the caller's FPR save area.
4. Save all nonvolatile GPRs that are used by this procedure in the caller's 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 8-19 of the CR if necessary.
5. If you saved any FPRs, restore them.
6. Return to caller.

---

## Stack Size Limit

Although the stack's storage limit is 64 MB, its size defaults to 512 KB. For applications that use large amounts of stack storage, you can set the limit to the maximum allowed. For example, you can use this **bash** command:

```
ulimit -s 65535
```

Some applications may exceed the allowed maximum. You can try compiling these applications with the **-qsave** compiler option, which defaults to the **STATIC** storage class.

---

## 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 176.

## 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.

**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`

**Symptom:** The shell cannot locate the command to execute the compiler.

**Solution:** Make sure that your `PATH` environment variable includes the directory `/opt/ibmcmp/xlF/8.1/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 the maximum allowed, or “unlimited”, respectively, for users who experience this problem. For example, you can set both your hard and soft limits with these `bash` commands:

```
ulimit -s 65535
ulimit -d unlimited
```

If the storage problem is in an XLF-compiled program, using the `-qsave` 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.

---

**Could not load program** *program*

**Could not load library** *library\_name.dylib*

**Error was: no such file or directory**

**Solution:** Make sure the XL Fortran dynamic libraries are installed in `/opt/ibmcmp/lib`, or set the `DYLD_LIBRARY_PATH` environment variable to include the directory where `libxlF90.dylib` is installed if it is in a different directory.

---

**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.

---

## Fixing Compile-Time Problems

The following sections discuss common problems you might encounter while compiling and how to avoid them.

### 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 53 for a list of these options and “Porting Programs to XL Fortran” on page 275 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.

## 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.

You may need to increase the system page space and recompile your program.

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.

---

## 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:

---

**/usr/bin/ld: Undefined symbols:**

**\_example\_missing\_function\_name**

**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 34 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 41.

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 53

on page 53 for a list of these options and “Porting Programs to XL Fortran” on page 275 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 Advanced Edition for Mac OS X 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 Advanced Edition for Mac OS X 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 this catalog is not installed on the system, only the message number appears. Some of the settings in “Setting Run-Time Options” on page 34 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 210 for instructions about exception handlers and “XL Fortran Run-Time Exceptions” on page 41 for information about the causes of run-time exceptions.

## Stack Size Overflows

The stack's storage limit is 64 MB. Some applications that use large amounts of stack storage may exceed this limit. See "Stack Size Limit" on page 258 for workaround suggestions.

---

## 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 49.



---

## 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.

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 51.

---

### 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.

---

### 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** option has 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

Except for the header, the above sections are repeated for each compilation unit when more than one compilation unit is present. The header occurs only once at the beginning 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 279.



---

## Software Development Topics

This section contains instructions and information on topics that are not directly related to XL Fortran features but may help you during the software development process.

- “Porting Programs to XL Fortran” on page 275



---

## 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 Advanced Edition for Mac OS X Language Reference*.
  - Some extensions require you to specify an XL Fortran compiler option; you can find these options listed in Table 7 on page 53.
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, but finding the correct solution may be time-consuming.

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 105.

---

## 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 Advanced Edition for Mac OS X 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>
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 ( <i>rtc</i> , <i>irtc</i> , <i>jdate</i> , <i>clock_</i> , <i>timef</i> , and <i>date</i> )	<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 or `EXTERNAL` 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 209.

## 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 136.



---

## Appendix A. Sample Fortran Programs

The following programs are provided as coding examples for XL Fortran. Other examples can be found in the `/opt/ibmcomp/xlf/8.1/samples/` directory. These illustrate various aspects of XL Fortran programming. 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 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

```

---

## 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. 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.

---

### 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

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.

---

## 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.

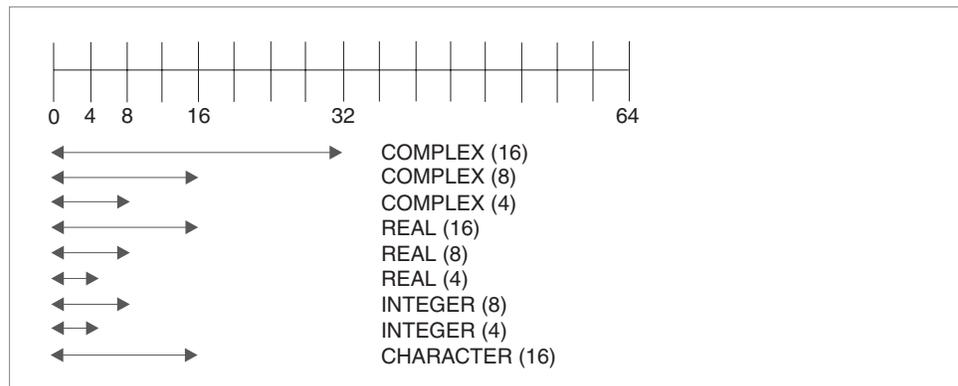


Figure 5. Storage Relationships without the -qautodbl Option

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
```

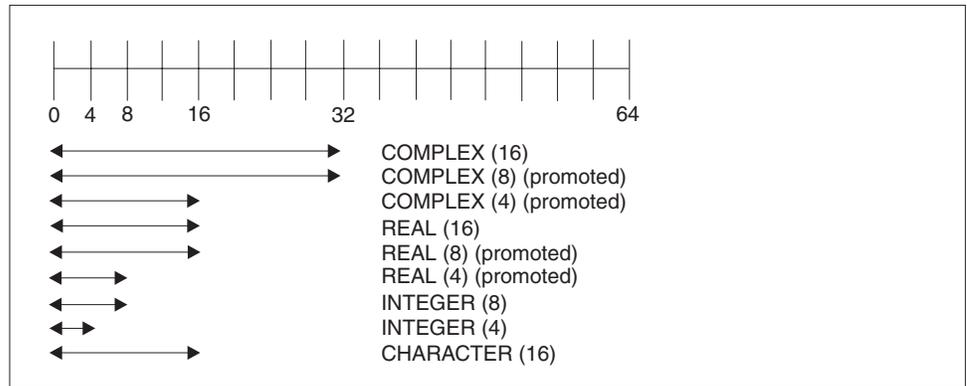


Figure 6. Storage Relationships with `-qautodbl=dbl`

```

@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

```

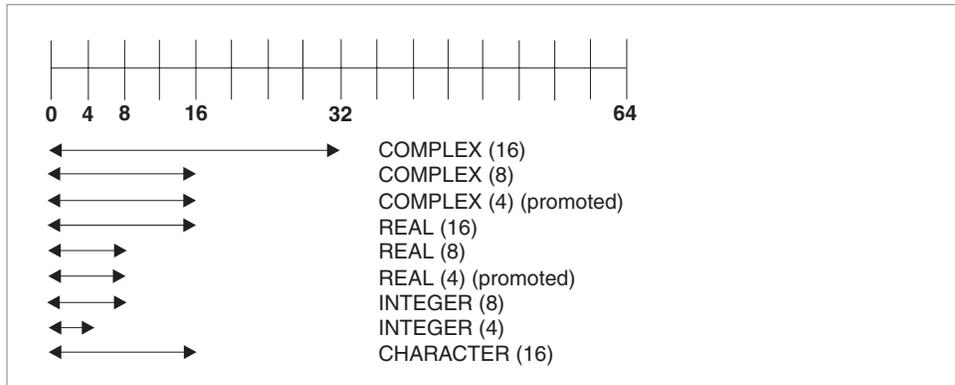


Figure 7. Storage Relationships with `-qautobl=dbl4`

```
@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
```

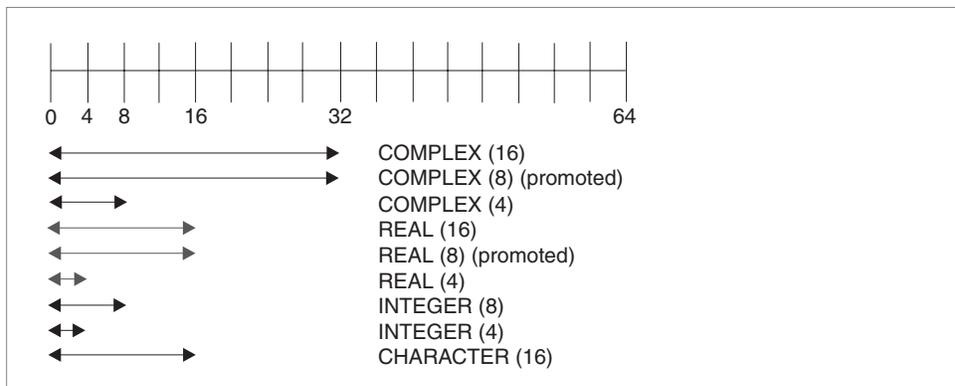


Figure 8. Storage Relationships with `-qautodbl=dbl8`

```
@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
```

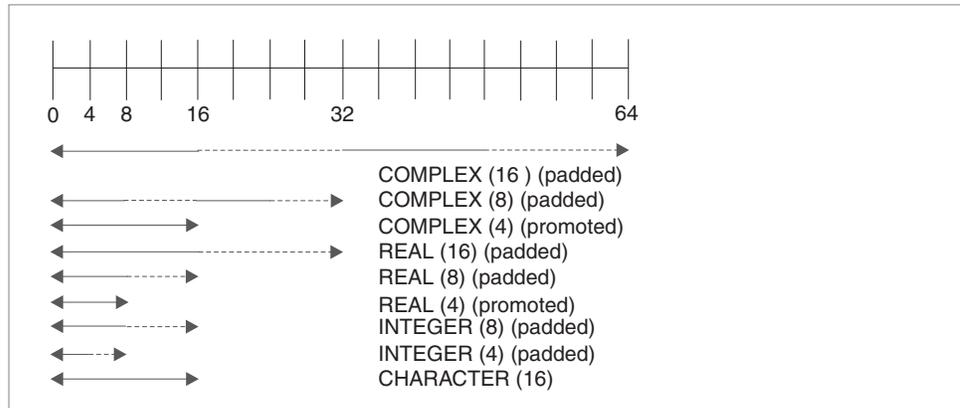


Figure 9. Storage Relationships with `-qautodbl=dblpad4`

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
```

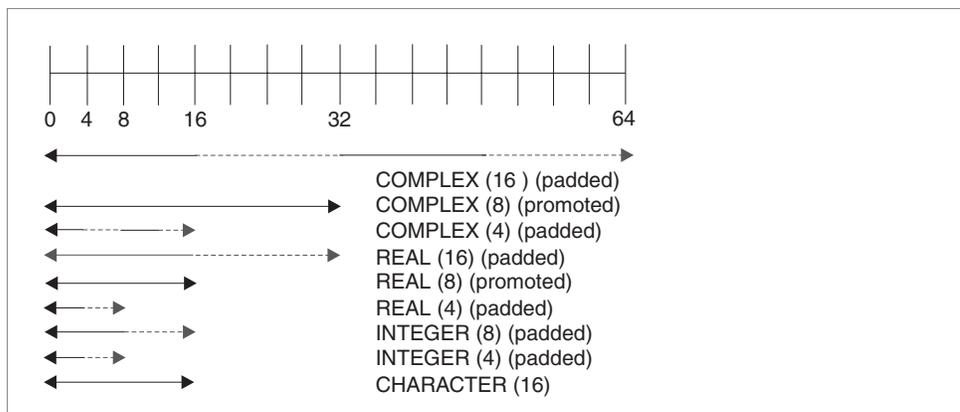


Figure 10. Storage Relationships with `-qautodbl=dblpad8`

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

```

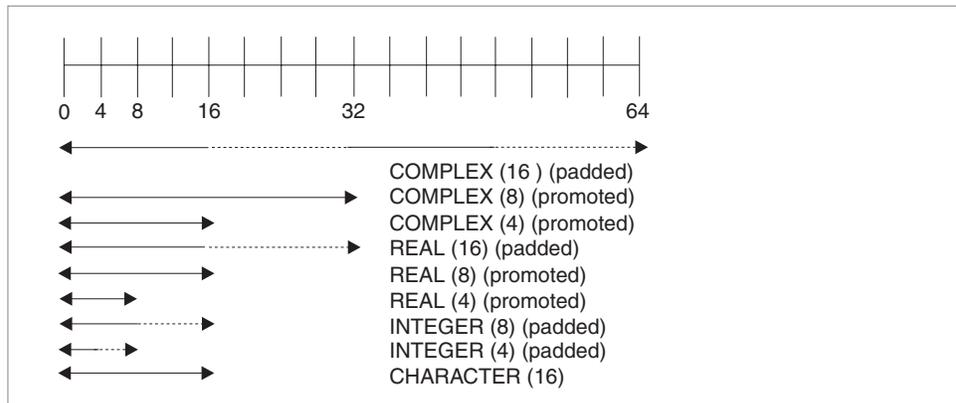


Figure 11. Storage Relationships with `-qautodbl=dblpad`

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. 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 <b>INTEGER(n)</b> for $n = 1, 2$ or $4$	$(2^{**31})-1$
Maximum number of iterations performed by <b>DO</b> loops with loop control with index variable of type <b>INTEGER(8)</b>	$(2^{**63})-1$
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	The record number can be up to $(2^{**63})-1$ . The maximum record length 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 $2^{**31}-1$ .
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 6700 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.

---

## 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:

AIX  
SAA

IBM  
z/OS

PowerPC

UNIX is a registered trademark of the Open Group in the United States and other countries.

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 book. It includes definitions that were developed by the American National Standards Institute (ANSI) and entries from the *IBM Dictionary of Computing*.

### A

**active processors.** See *online processors*.

**alias.** An alternative label for a data object or point in a computer program.

**American National Standards Institute (ANSI).** An organization consisting of producers, consumers, and general interest groups, that establishes the procedures by which accredited organizations create and maintain voluntary industry standards in the United States.

**asynchronous.** Not synchronized in time. For example, input events are controlled by the user; the program can read them later.

### B

**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.

### C

**chunk.** A subset of consecutive loop iterations.

**compile.** To translate a program that is written in a high-level programming language into a machine language program. The program that performs this task is called a *compiler*.

### 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 type.** The properties and internal representation that characterize data and functions.

**denormalized number.** An IEEE number with a very small absolute value and lowered precision. Denormal numbers are represented by a zero exponent and a non-zero fraction.

**dynamic dimensioning.** The process of re-evaluating the bounds of a pointer array each time the pointer is referenced.

### E

**environment variable.** A variable that describes the operating environment of the process.

**executable program.** A program that can be run. It consists of a main program, and, optionally, one or more subprograms or non-Fortran-defined external procedures, or both.

**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.

### F

**floating-point number.** A real number that is 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.

**Fortran (Formula Translation).** A high-level programming language that is used primarily for scientific, engineering, and mathematical applications.

**function.** A procedure that returns the value of a single variable and that usually has a single exit.

### 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.

**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.

## 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.

**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. Synonym for file index.

**intrinsic.** An adjective applied 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 procedures.** Fortran defines a number of procedures, called intrinsic procedures, that are available to any program.

## L

**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.

**load balancing.** An optimization strategy that aims at evenly distributing the work load among processors.

## 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.

## N

**NaN.** See not-a-number.

**not-a-number (NaN).** A symbolic entity encoded in floating-point format. There are two types of NaNs. Signalling NaNs signal the invalid operation exception whenever they appear as operands. Quiet NaNs propagate through almost every arithmetic operation without signaling exceptions. Both types of NaNs represent anything that is not a number. The intent of the signaling NaN is to catch program errors, such as

using an uninitialized variable. The intent of a quiet NaN is to propagate a NaN result through subsequent computations.

## O

**online processors.** Also known as *active processors*. In a multiprocessor machine, this refers to the processors which the system administrator has decided to activate (bring online). This number is less than or equal to the number of physical processors actually installed in the machine.

**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.)

## P

**paging space.** Disk storage for information that is resident in virtual memory but is not currently being accessed.

**PDF.** Profile-directed feedback, a type of optimization that uses information collected during application execution to improve performance of conditional branches and in frequently executed sections of code.

**pointee array.** Explicit-shape or assumed-size arrays that are declared in **INTEGER** pointer statements or other specification statements.

**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.

## S

**semantics.** The relationships of characters or groups of characters to their meanings, independent of the manner of their interpretation and use. Contrast with *syntax*.

**sleep.** The state in which a thread completely suspends execution until another thread signals it that there is work to do.

**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.

**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.

**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.

**subroutine.** A procedure that is invoked by a **CALL** statement or by a defined assignment statement.

**synchronously.** The way in which signals caused by interrupts are generated.

**syntax.** The rules for the construction of a statement. Contrast with *semantics*.

## T

**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.

**trigger constant.** A sequences of characters that identifies comment lines as compiler comment directives.

## U

**Unicode.** The informal name for the Universal Coded Character set (UCS), which is the name of the ISO 10646 standard that defines a single code for the representation, interchange, processing, storage, entry, and presentation of the written form of the world's major languages.

**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.

## 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.



# INDEX

## Special characters

# compiler option 63  
-l compiler option 64  
-B compiler option 65  
-c compiler option 67  
-C compiler option 66  
-d compiler option 69  
-D compiler option 68  
-F compiler option 70  
-g compiler option 71, 265  
-I compiler option 72  
-k compiler option 73  
-l compiler option 75  
-L compiler option 74  
-N compiler option 76  
-o compiler option 79  
-O compiler option 77, 219  
-O2 compiler option 77  
-O3 compiler option 77  
-O4 compiler option 77  
-O5 compiler option 78  
-p compiler option 80  
-qalias compiler option 81, 223  
-qalign compiler option 84  
-qarch compiler option 30, 86, 221  
-qassert compiler option 88, 223  
-qattr compiler option 89, 270  
-qautodbl compiler option 90, 284  
-qcache compiler option 30, 92, 221  
-qcclines compiler option 94  
-qcharlen compiler option 95  
-qcheck compiler option 66, 96  
-qci compiler option 97  
-qcommon compiler option 98  
-qcompact compiler option 99  
-qcr compiler option 100  
-qctypless compiler option 101  
-qdbg compiler option 71, 103  
-qddim compiler option 104  
-qdirective compiler option 105  
-qdlines compiler option 68, 107  
-qdpc compiler option 108  
-qescape compiler option 109  
-qextern compiler option 110  
-qextname compiler option 111  
-qfixed compiler option 113  
-qflag compiler option 114  
-qfloat compiler option 115, 209  
    fltint suboption 209  
    nans suboption 215  
    nomaf suboption 209  
    rsqrt suboption 209  
-qflttrap compiler option 117, 210  
-qfree compiler option 119  
-qfullpath compiler option 120  
-qhalt compiler option 121  
-qhot compiler option 122, 223  
-qieee compiler option 123, 199  
-qinit compiler option 124  
-qinitauto compiler option 125  
-qintlog compiler option 127  
-qintsize compiler option 128  
-qipac compiler option 130, 229  
-qkeepparm compiler option 135  
-qlanglvl compiler option 136  
-qlibansi linker option 133  
-qlibposix linker option 133  
-qlist compiler option 140, 271  
-qlistopt compiler option 141, 267  
-qlog4 compiler option 142  
-qmaxmem compiler option 143  
-qmbcs compiler option 145  
-qmixed compiler option 146  
-qmoddir compiler option 147  
-qnoprint compiler option 148  
-qnulterm compiler option 149  
-qobject compiler option 150  
-qonetrip compiler option 64, 151  
-qoptimize compiler option 77, 152  
-qpdf compiler option 153, 226  
-qpghsinfo compiler option 156  
-qpvc compiler option 157  
-qport compiler option 158  
-qposition compiler option 159, 235  
-qprefetch compiler option 160  
-qqcount compiler option 161  
-qrealize compiler option 162  
-qrecur compiler option 164  
-qreport compiler option 165, 269  
-qsaac compiler option 166  
-qsave compiler option 167  
-qsigtrap compiler option 168, 210  
-qsmallstack compiler option 169  
-qsource compiler option 170, 268  
-qspillsize compiler option 76, 171  
-qstrict compiler option 172, 219  
-qstrict\_induction compiler option 174  
-qstrictieemod compiler option 173  
-qsuffix compiler option 175  
-qsuppress compiler option 176  
-qthreaded compiler option 178  
-qtune compiler option 30, 179, 221  
-qundef compiler option 180, 194  
-qunroll compiler option 181  
-qunwind compiler option 182  
-qxflag=oldtab compiler option 183  
-qxl77 compiler option 184  
-qxl90 compiler option 186  
-qxlines compiler option 188  
-qxref compiler option 190, 270  
-qzerosize compiler option 191  
-t compiler option 192  
-u compiler option 194  
-U compiler option 193  
-v compiler option 195  
-V compiler option 196  
-w compiler option 114, 198  
-W compiler option 197  
-yn, -ym, -yp, -yz compiler options 123, 199  
/etc/opt/ibmcomp/xf/8.1/xf.cfg  
    configuration file 15, 70

/tmp directory  
    See TMPDIR environment variable  
.a files 25  
.cfg files 25  
.dylib files 25  
.f and .F files 25  
.f90 suffix, compiling files with 15  
.lst files 26  
.mod files 25, 26, 147  
.o files 25, 26  
.s files 25, 26  
.XOR. operator 184  
\* length specifiers (FORTRAN 77  
    extension) 276  
@PROCESS compiler directive 29  
%REF functions 248  
%VAL functions 248  
#if and other cpp directives 31

## Numerics

1501-229, and 1517-011 error  
    messages 263  
15xx identifiers for XL Fortran  
    messages 260  
4K suboption of -qalign 84  
64-bit data types (FORTRAN 77  
    extension) 276

## A

a.out file 26  
addresses of arguments, saving 184  
ALIAS @PROCESS directive 81  
ALIGN @PROCESS directive 84  
alignment of CSECTs and large arrays for  
    data-striped I/O 84  
allocatable arrays, automatic deallocation  
    with -qxlf90=autodealloc 186  
ANSI  
    checking conformance to the Fortran  
        90 standard 10, 38, 136  
    checking conformance to the Fortran  
        95 standard 10, 38, 136  
    definition of 297  
appendold and appendunknown  
    suboptions of -qposition 159  
archive files 25  
argument addresses, saving 184  
argument promotion (integer only) for  
    intrinsic procedures 184  
arguments  
    passing between languages 244, 245  
    passing by reference or by value 248  
    passing null-terminated strings to C  
        functions 149  
arraypad suboption of -qhot 225  
arrays  
    optimizing array language 223  
    optimizing assignments 81

arrays (*continued*)  
 passing between languages 247  
 aryovrlp suboption of -qalias 81, 223  
 as and asopt attributes of configuration file 15  
 as command, passing command-line options to 29  
 assembler  
 low-level linkage conventions 251  
 source (.s) files 25  
 ATTR @PROCESS directive 89  
 attribute section in compiler listing 270  
 auto suboption of -qarch 86  
 auto suboption of -qipaa 130  
 auto suboption of -qtune 179  
 AUTODBL @PROCESS directive 90  
 autodealloc suboption of -qxl90 186

## B

bash shell 13  
 bitwise-identical floating-point results 209  
 blankpad suboption of -qxl77 184  
 blocked special files, interaction of XL Fortran I/O with 236  
 bolt attribute of configuration file 15  
 branches, optimizing 226  
 bss storage, alignment of arrays in 84  
 buffering run-time option  
 description 35  
 using with preconnected files 35  
 buffers, flushing 238  
 BYTE data type (FORTRAN 77 extension) 276

## C

C language and interlanguage calls 241, 244  
 C preprocessor (cpp) 31  
 C++ and Fortran in same program 242  
 calling by reference or value 248  
 calling non-Fortran procedures 241  
 carriage return character 100  
 CCLINES @PROCESS 94  
 character constants and typeless constants 101  
 character data, passing between languages 246  
 character special files, interaction of XL Fortran I/O with 236  
 character-count edit descriptor (FORTRAN 77 extension) 276  
 CHARLEN @PROCESS directive 95  
 CHECK @PROCESS directive 66, 96  
 check\_fpscr.f sample file 214  
 CI @PROCESS directive 97  
 cleanpdf command 155  
 cnvrr run-time option 37  
 code attribute of configuration file 15  
 code generation for different systems 30  
 code optimization 11, 217  
 command line, specifying options on 28

command-line options  
 See compiler options  
 COMMON @PROCESS directive 98  
 COMPACT @PROCESS directive 99  
 compexgcc suboption of -qfloat 115  
 compilation order 25  
 compilation unit epilogue section in compiler listing 271  
 compiler listings 267  
 See also listings  
 compiler options for controlling 51  
 compiler options  
 See also the individual options listed under Special Characters at the start of the index  
 deprecated 61  
 descriptions 62  
 for compatibility 53  
 for controlling input to the compiler 44  
 for controlling listings and messages 51  
 for controlling the compiler internal operation 60  
 for debugging and error checking 49, 50  
 for floating-point processing 59  
 for linking 59  
 for new language extensions 58  
 for performance optimization 46  
 obsolete or not recommended 61  
 scope and precedence 27  
 section in compiler listing 267  
 specifying in the source file 29  
 specifying on the command line 28  
 specifying the locations of output files 45  
 summary 43  
 compiling  
 cancelling a compilation 25  
 description of how to compile a program 23  
 problems 262  
 conditional branching optimization 226  
 conditional compilation 31  
 conditional vector merge intrinsic functions (FORTRAN 77 extension) 276  
 configuration file 15, 25, 70  
 conflicting options  
 -C interferes with -qhot 66  
 -qautodbl overrides -qrealsize 91  
 -qdpcc is overridden by -qautodbl and -qrealsize 162  
 -qflag overrides -qlanglvl and -qsa 114  
 -qhalt is overridden by  
 -qnoobject 150  
 -qhalt overrides -qobject 150  
 -qhot is overridden by -C 122  
 -qintsize overrides -qlog4 142  
 -qlanglvl is overridden by -qflag 137  
 -qlog4 is overridden by -qintsize 142  
 -qnoobject overrides -qhalt 121  
 -qobject is overridden by -qhalt 121  
 -qrealsize is overridden by -qautodbl 91, 163

conflicting options (*continued*)  
 -qrealsize overrides -qdpcc 162  
 -qsa is overridden by -qflag 166  
 @PROCESS overrides command-line setting 27  
 command-line overrides configuration file setting 27  
 specified more than once, last one takes effect 28  
 conformance checking 10, 136, 166  
 control and status register for floating point 213  
 conversion errors 37  
 core file 210, 264  
 cost model for loop transformations 223  
 could not load program (error message) 262  
 cpp command 31  
 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 276  
 date and time service and utility functions 276  
 CRAY pointer (FORTRAN 77 extension), XL Fortran equivalent 276  
 cross-reference section in compiler listing 270  
 crt attribute of configuration file 15  
 CSECTS, alignment of 84  
 csh shell 13  
 CTYPLSS @PROCESS directive 101  
 customizing configuration file (including default compiler options) 15  
 CVMGx intrinsic functions (FORTRAN 77 extension) 276

## D

data limit 262  
 data stripping  
 -qalign required for improved performance 84  
 data types in Fortran, C 245  
 date and time functions (FORTRAN 77 extension) 276  
 DBG @PROCESS directive 71, 103  
 dbl, dbl4, dbl8, dblpad, dblpad4, dblpad8 suboptions of -qautodbl 90  
 DDIM @PROCESS directive 104  
 debugger support 11  
 debugging 259  
 compiler options for 49  
 using path names of original files 120  
 defaultmsg attribute of configuration file 15  
 defaults  
 customizing compiler defaults 15  
 search paths for include and .mod files 72  
 deprecated compiler options 61  
 deps suboption of -qassert 88  
 DIRECTIVE @PROCESS directive 105  
 disk space, running out of 263

disk striping  
   *See* data striping  
 DLINES @PROCESS directive 68, 107  
 documentation, online formats 11  
 double-precision values 202, 204  
 DPC @PROCESS directive 108  
 DYLD\_LIBRARY\_PATH 41  
 dynamic dimensioning of arrays 104  
 dynamic linking 33

**E**

E error severity 259  
 edit descriptors (B, O, Z), differences  
   between F77 and F90 184  
 edit descriptors (G), difference between  
   F77 and F90 184  
 editing source files 23  
 emacs text editor 23  
 enable suboption of -qfltrap 117, 212  
 end-of-file, writing past 184  
 ENTRY statements, compatibility with  
   previous compiler versions 184  
 environment problems 262  
 environment variables  
   compile time 13  
     PDFDIR 14  
     TMPDIR 14  
   run-time  
     PDFDIR 14  
     TMPDIR 41  
     XLFRTEOPTS 34  
     XLFSCRATCH\_unit 14  
     XLFUNIT\_unit 14  
 eof, writing past 184  
 epilogue sections in compiler listing 271  
 err\_recovery run-time option 38  
 error checking, compiler options for 49  
 error messages 259  
   1501-229 263  
   1517-011 263  
   compiler options for controlling 51  
   explanation of format 260  
   in compiler listing 268  
 erroreof run-time option 38  
 ESCAPE @PROCESS directive 109  
 example programs  
   *See* sample programs  
 exception handling 41, 203  
   for floating point 117, 209  
   installing an exception handler 210  
 exclusive or operator 184  
 executable files 26  
 executing a program 34  
 executing the compiler 23  
 exits suboption of -qipa 130  
 explicit interfaces 249  
 extended-precision values 205  
 extensions to FORTRAN 77, list of  
   common ones 276  
 external names  
   in the run-time environment 284  
 EXTNAME @PROCESS directive 111

**F**

f77 command  
   description 23  
   level of Fortran standard  
     compliance 24  
 f90 suffix 15  
 fexcp.h include file 211  
 fhandler.F sample file 214  
 file positioning 235  
 file table section in compiler listing 271  
 files  
   editing source 23  
   I/O formats 233  
   input 25  
   names 234  
   output 26  
   permissions 237  
   using suffixes other than .f for source  
     files 15  
 FIPS FORTRAN standard, checking  
   conformance to 10  
 FIXED @PROCESS directive 113  
 FLAG @PROCESS directive 114  
 FLOAT @PROCESS directive 115  
 floating-point  
   exception handling 41  
   exceptions 117, 209  
   processing 201  
     optimizing 209, 222  
 floating-point status and control  
   register 213  
 flint suboption of -qfloat 115  
 FLTRAP @PROCESS directive 117, 210  
 fltrap\_handler.c and fltrap\_test.f sample  
   files 214  
 flushing I/O buffers 238  
 fold suboption of -qfloat 115  
 formats, file 233  
 fort.\* default file names 234, 238  
 fort77 command  
   description 23  
 Fortran  
   compiler options for language  
     extensions 58  
 FORTRAN 77 extensions, list of common  
   ones 276  
 Fortran 90  
   compiling programs written for 24  
   fpdt.h and fpdc.h include files 206  
   fpgets and fpsets service and utility  
     subroutines 213  
   fppv and fppk attributes of configuration  
     file 15  
   fpscr register 213  
   fpstat array 213  
 FREE @PROCESS directive 119  
 fsuffix attribute of configuration file 15  
 FULLPATH @PROCESS directive 120  
 functions  
   linkage convention for calls 256  
   return values 250

**G**

G edit descriptor, difference between F77  
 and F90 184

g5 suboption of -qarch 86  
 g5 suboption of -qtune 179  
 gcr attribute of configuration file 15  
 gedit77 suboption of -qxl77 184  
 generating code for different systems 30  
 get\_round\_mode procedure 206  
 GETENV intrinsic procedure 234

**H**

HALT @PROCESS directive 121  
 hardware, compiling for different types  
   of 30  
 header section in compiler listing 267  
 hexint and nohexint suboptions of  
   -qport 158  
 hot attribute of configuration file 15  
 hotlist suboption of -qreport 165  
 HTML documentation 11

**I**

I error severity 259  
 i-node 39  
 I/O  
   *See* input/output  
 IBM Distributed Debugger 11  
 IEEE @PROCESS directive 123, 199  
 IEEE arithmetic 201  
 implicitly connected files 234  
 imprecise suboption of -qfltrap 117  
 include files fpdt.h and fpdc.h 206  
 include\_32 attribute of configuration  
   file 15  
 inexact suboption of -qfltrap 117  
 infinity values 202  
 informational message 259  
 INIT @PROCESS directive 124  
 initial file position 235  
 inline suboption of -qipa 130  
 inlining 228  
 input files 25  
 input/output 201  
   from two languages in the same  
     program 242  
   increasing throughput with data  
     striping 84  
   redirection 236  
   run-time behavior 34  
   when unit is positioned at  
     end-of-file 184  
   XL Fortran implementation  
     details 233  
 installation problems 262  
 installing the compiler 13  
 intarg suboption of -qxl77 184  
 integer arguments of different kinds to  
   intrinsic procedures 184  
 integer POINTER (FORTRAN 77  
   extension) 276  
 INTENT attribute 250  
 interlanguage calls 241, 248  
   arrays 247  
   C++ 242  
   character types 246  
   corresponding data types 245

- interlanguage calls (*continued*)
  - input and output 242
  - low-level linkage conventions 251
  - pointers 248
- internal limits for the compiler 291
- interprocedural analysis (IPA) 130
- INTLOG @PROCESS directive 127
- intptr suboption of -qalias 81
- intrinsic procedures accepting integer arguments of different kinds 184
- intrinths run-time option 38
- INTSIZE @PROCESS directive 128
- intxor suboption of -qxf77 184
- invalid suboption of -qfltrap 117
- invoking a program 34
- invoking the compiler 23
- ipa attribute of configuration file 15
- irand routine, naming restriction for 33
- ISO
  - checking conformance to the Fortran 90 standard 10, 38, 136
  - checking conformance to the Fortran 95 standard 10, 38, 136
- isolated suboption of -qipa 130
- itercnt suboption of -qassert 88

## K

- kind type parameters 128, 162
- ksh shell 13

## L

- L error severity 259
- LANGLVL @PROCESS directive 136
- langlvl run-time option 38
- language support 9
- language-level error 259
- ld and ldopt attributes of configuration file 15
- ld command
  - passing command-line options to 29
- leadzero suboption of -qxf77 184
- level suboption of -qipa 130
- lib\*.dylib 25
- lib\*.dylib library files 75
- libraries 25
  - shared 283
- libraries attribute of configuration file 15
- library path environment variable 262
- libxf90\_r.dylib 24
- libxf90\_r.dylib library 19
- libxf90\_t.dylib 24
- libxf90\_t.dylib library 19
- limit command 262
- limit suboption of -qipa 130, 229
- limits internal to the compiler 291
- line feed character 100
- linker options 59
  - qlibansi 133
  - qlibposix 133
- linking 32
  - dynamic 33
  - problems 263

- links, interaction of XL Fortran I/O with 236
- LIST @PROCESS directive 140
- list suboption of -qipa 130, 131
- listing files 26
- listing options 51
- LISTOPT @PROCESS directive 141
- LOG4 @PROCESS directive 142
- long variable names (FORTRAN 77 extension) 276
- loops, optimizing 223
- lower case (FORTRAN 77 extension) 276
- lowfreq suboption of -qipa 130

## M

- m suboption of -y 199
- machines, compiling for different types 30, 86
- macro expansion 31
- maf suboption of -qfloat 115, 172
- main, restriction on use as a Fortran name 241
- make command 63
- makefiles
  - configuration file as alternative for default options 15
  - copying modified configuration files along with 15
- malloc system routine 91
- MAXMEM @PROCESS directive 143
- MBCS @PROCESS directive 145
- mclock routine, naming restrictions for 33
- memory management optimizations 223
- message suppression 176
- messages
  - 1501-229 error message 263
  - 1517-011 error message 263
  - catalog files for 261
  - compiler options for controlling 51
  - copying message catalogs to another system 261
- migrating 9
  - from other systems 275
- minus infinity, representation of 202
- minus suboption of -qieee 123
- missing suboption of -qipa 130
- MIXED @PROCESS directive 146, 193
- mixing integers and logicals (FORTRAN 77 extension) 276
- mod and nomod suboptions of -qport 158
- mod files 25, 26, 147
- module procedures, external names corresponding to 241
- modules, effect on compilation order 25
- mon.out file 25
- multconn run-time option 39
- multconnio run-time option 39

## N

- n suboption of -y 199
- name conflicts, avoiding 33
- namelist run-time option 40

- naming conventions for external names 241
- NaN values
  - and infinities 202
  - specifying with -qinitauto compiler option 125
- nans suboption of -qfloat 115
- nearest suboption of -qieee 123
- negative infinity, representation of 202
- nlwidth run-time option 40
- nodblpad suboption of -qautodbl
  - See none suboption instead
- nodeps suboption of -qassert 88
- noinline suboption of -qipa 130
- none suboption of -qautodbl 90
- noobject suboption of -qipa 130
- null-terminated strings, passing to C functions 149, 246
- NULLTERM @PROCESS directive 149

## O

- OBJECT @PROCESS directive 150
- object files 25, 26
- object suboption of -qipa 130
- obsolete compiler options 61
- oldboz suboption of -qxf77 184
- ONETRIP @PROCESS directive 64, 151
- online compiler help 11
- online documentation 11
- optimization 11, 217
  - compiler options for 46
  - for floating-point arithmetic 209
  - levels 219
- OPTIMIZE @PROCESS directive 77, 152
- OPTIONAL attribute 250
- options attribute of configuration file 15
- options section in compiler listing 267
- osuffix attribute of configuration file 15
- output files 26
- overflow suboption of -qfltrap 117

## P

- p suboption of -y 199
- pad setting, changing for internal, direct-access and stream-access files 184
- padding of data types with -qautodbl option 284
- paging space
  - running out of 263
- parameters
  - See arguments
- partition suboption of -qipa 130
- Pascal language and interlanguage calls 241
- path name of source files, preserving with -qfullpath 120
- PDF documentation 11
- PDFDIR environment variable 14
- pdfname suboption of -qipa 130, 132
- performance of floating-point arithmetic 209
- performance of real operations, speeding up 91, 162

- permissions of files 237
- persistent suboption of `-qxlf77` 184
- PHSINFO @PROCESS directive 156
- pipes, interaction of XL Fortran I/O with 236
- platform, compiling for a specific type 86
- plus infinity, representation of 202
- plus suboption of `-qiee` 123
- pointers (Fortran 90) and `-qinit` compiler option 124
- pointers (integer POINTER) (FORTRAN 77 extension) 276
- PORT @PROCESS directive 158
- portability 275
- porting to XL Fortran 275
- POSITION @PROCESS directive 159, 235
- position of a file after an OPEN statement 235
- positive infinity, representation of 202
- postmortem.f sample file 214
- PowerPC systems
  - compiling programs for 30
  - ppc970 suboption of `-qarch` 86
  - ppc970 suboption of `-qtune` 179
  - ppcv suboption of `-qarch` 86
- precision of real data types 91, 162
- preconnected files 234
- preprocessing Fortran source with the C preprocessor 31
- problem determination 259
- procedure calls to other languages
  - See* subprograms in other languages, calling
- prof command 26
- profiling data files 26
- Program Editor 23
- Project Builder 19
- Projects, sample 19
- promoting integer arguments to intrinsic procedures 184
- promotion of data types with `-qautodbl` option 284
- pseudo-devices, interaction of XL Fortran I/O with 236
- pteovrlp suboption of `-qalias` 81
- pure suboption of `-qipa` 130

## Q

- Q (character-count) edit descriptor (FORTRAN 77 extension) 276
- QCOUNT @PROCESS directive 161
- quiet NaN 125, 202

## R

- rand routine, naming restriction for 33
- random run-time option 40
- READ statements past end-of-file 184
- README.xlf file 13
- real arithmetic 201
- REAL data types 91
- REAL(16) values 205
- REAL(4) and REAL(8) values 202, 204

- REALSIZE @PROCESS directive 162
- record lengths 237
- RECUR @PROCESS directive 164
- recursion 164, 167
- redirecting input/output 236
- reference, passing arguments by 248
- register flushing 135
- related documentation 6
- REPORT @PROCESS directive 165
- resetpdf command 155
- return code
  - from compiler 260
  - from Fortran programs 260
- rounding 206
  - rounding errors 208
  - rounding mode 206, 208
- rrm suboption of `-qfloat` 115, 172
- rsqrt suboption of `-qfloat` 115
- run time
  - exceptions 41
  - options 34
- run-time
  - libraries 25
  - problems 263
- run-time environment
  - external names in 284
- running a program 34
- running the compiler 23

## S

- S error severity 259
- SAA @PROCESS directive 166
- SAA FORTRAN definition, checking conformance to 10
- safe suboption of `-qipa` 130
- sample programs 279
  - calling C functions from Fortran 246
  - floating-point exception handling 214
  - notes on using 6
- SAVE @PROCESS directive 167
- scratch file directory
  - See* TMPDIR environment variable
- scratch\_vars run-time option 14, 40, 239
- segmentation fault 153
- setrteopts service and utility procedure 34
- severe error 259
- sh shell 13
- shared libraries 283
- shared object files 25
- side-effects, definition of 130
- SIGFPE signal 210
- SIGN intrinsic, effect of `-qxlf90=signedzero` on 186
- signal handling 41
  - for floating point 209
  - installing an exception handler 210
- signaling NaN 202, 215
- signedzero suboption of `-qxlf90` 186
- SIGTRAP signal 41, 210
- single-precision values 202, 204
- softeof suboption of `-qxlf77` 184
- SOURCE @PROCESS directive 170
- source file options 29
- source files 25

- source files (*continued*)
  - allowing suffixes other than `.f` 15
  - preserving path names for debugging 120
  - specifying options in 29
- source section in compiler listing 268
- source-code conformance checking 10
- source-level debugging support 11
- space problems 262
- special files, interaction of XL Fortran I/O with 236
- SPILLSIZE @PROCESS directive 76, 171
- ssuffix attribute of configuration file 15
- stack 252
  - limit 262
- standard error, input, and output streams 234
- star length specifiers 276
- static storage, alignment of arrays in 84
- status and control register for floating point 213
- std suboption of `-qalias` 81
- stderr, stdin, and stdout streams 234
- stdexits suboption of `-qipa` 130
- storage limits 262
- storage relationship between data objects 284
- storage-associated arrays, performance implications of 81
- STRICT @PROCESS directive 172
- strictieemod @PROCESS directive 173
- strictnmaf suboption of `-qfloat` 115
- strings, passing to C functions 149, 246
- subprogram calls to other languages
  - See* subprograms in other languages, calling
- subprograms in other languages, calling 241, 244
- suffix, allowing other than `.f` on source files 15
- suffixes for source files 175
- summary of compiler options 43
- Sun pointer (FORTRAN 77 extension), XL Fortran equivalent 276
- symbolic debugger support 11
- symbolic links, interaction of XL Fortran I/O with 236
- syntax diagrams and statements 4
- system problems 262

## T

- target machine, compiling for 86
- tctl command 236
- temporary arrays, reducing 81, 223
- temporary file directory 14
- temporary files
  - See* /tmp directory
- text editors 23
- TextEdit text editor 23
- threads, controlling 38
- threshold suboption of `-qipa` 130
- throughput for I/O, increasing with data striping 84
- time and date functions (FORTRAN 77 extension) 276
- times routine, naming restriction for 33

- TMPDIR environment variable 41, 263
  - compile time 14
- Trace/BPT trap 41, 210
- traceback listing 168, 211, 264
- transformation report section in compiler listing 269
- trigger\_constant
  - IBM\* 105
  - IBMT 178
  - setting values 105
- trigraphs 32
- tuning performance
  - See optimization
- typeless constants (FORTRAN 77 extension) 276
- typeless constants and character constants 101
- typstmt and notypstmt suboptions of -qport 158

## U

- U error severity 259
- ulimit command 262
- UNDEF @PROCESS directive 180, 194
- underflow suboption of -qflttrap 117
- Unicode data 145
- unit\_vars run-time option 14, 41, 238
- UNIVERSAL setting for locale 145
- unknown suboption of -qipa 130
- unrecoverable error 259
- unrolling DO LOOPS 181
- unrolling loops 224
- UNWIND @PROCESS directive 182
- use attribute of configuration file 15
- UTF-8 encoding for Unicode data 145

## V

- value relationships between data objects 284
- value, passing arguments by 248
- vi text editor 23

## W

- W error severity 259
- warning error 259
- WRITE statements past end-of-file 184

## X

- Xcode 19
- XFLAG(OLDTAB) @PROCESS directive 183
- xl\_ieee exception handler 211
- xl\_ieee.F and xl\_ieee.c sample files 214
- xl\_sigdump exception handler 211
- xl\_trbk exception handler 211
- xl\_trbk library procedure 264
- xl\_trbk\_test.f sample file 214
- xl\_trce exception handler 168, 211
- xl\_trcedump exception handler 211
- xl attribute of configuration file 15

- xl command
  - description 23
  - level of Fortran standard compliance 24
- xl\_r command
  - description 23
  - level of Fortran standard compliance 24
- xl.cfg configuration file 70
- XL77 @PROCESS directive 184
- XL90 @PROCESS directive 186
- xl90 command
  - description 23
  - level of Fortran standard compliance 24
- xl90\_r command
  - description 23
  - level of Fortran standard compliance 24
- xl95 command
  - description 23
- xl95\_r command
  - description 23
  - level of Fortran standard compliance 24
- xlfopt attribute of configuration file 15
- XLFRTEOPTS environment variable 34
- XLFSCRATCH\_unit environment variable 14, 40, 239
- XLUNIT\_unit environment variable 14, 41, 238
- XLINES @PROCESS 188
- XOR 184
- XREF @PROCESS directive 190
- xrf\_messages run-time option 41

## Z

- z suboption of -y 199
- zero suboption of -qiee 123
- zerodivide suboption of -qflttrap 117
- zeros (leading), in output 184
- ZEROSIZE @PROCESS directive 191





Program Number: 5724-G13

SC09-7864-00

