

IBM XL Fortran for Linux, V13.1



Compiler Reference

Version 13.1

IBM XL Fortran for Linux, V13.1



Compiler Reference

Version 13.1

Note

Before using this information and the product it supports, read the information in "Notices" on page 337.

First edition

This edition applies to IBM XL Fortran for Linux, V13.1 (Program 5724-X16) and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

© Copyright IBM Corporation 1990, 2010.

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

Contents

About this information vii

Who should read this information.	vii
How to use this information	vii
How this information is organized	vii
Conventions	viii
Related information	xii
IBM XL Fortran information.	xii
Standards and specifications	xiii
Other IBM information	xiv
Technical support	xiv
How to send your comments	xiv

Chapter 1. Introduction 1

Chapter 2. Overview of XL Fortran features 3

Hardware and operating-system support	3
Language support	3
Migration support	4
Source-code conformance checking	4
Highly configurable compiler.	4
Diagnostic listings	5
Symbolic debugger support	6
Program optimization	6

Chapter 3. Setting up and customizing XL Fortran. 7

Where to find installation instructions	7
Using the compiler on a network file System	7
Correct settings for environment variables	8
Environment variable basics	8
Environment variables for national language support	8
Setting library search paths	9
PDF environment variables	9
TMPDIR: Specifying a directory for temporary files	10
XLFSRATCH_unit: Specifying names for scratch files	10
XLFUNIT_unit: Specifying names for implicitly connected files	10
Using custom compiler configuration files	10
Creating custom configuration files	11
Editing the default configuration file	14
Attributes	14
Determining which level of XL Fortran is installed	16
Backward compatibility issues	17
Running two levels of XL Fortran	19

Chapter 4. Editing, compiling, linking, and running XL Fortran programs 21

Editing XL Fortran source files	21
Compiling XL Fortran programs	21
Compiling Fortran 90 or Fortran 95 programs	23

Compiling Fortran 2003 programs	23
Compiling and linking a library	24
Compiling XL Fortran SMP programs	25
Compilation order for Fortran programs.	25
Canceling a compilation	25
XL Fortran input files	25
XL Fortran output files	27
Scope and precedence of option settings.	28
Specifying options on the command line.	28
Specifying options in the source file	29
Passing command-line options to the "ld" or "as" command	30
Displaying information inside binary files (strings)	31
Compiling for specific architectures	31
Passing Fortran files through the C preprocessor	31
cpp directives for XL Fortran programs	33
Passing options to the C preprocessor	33
Avoiding preprocessing problems	33
Linking XL Fortran programs	33
Compiling and linking in separate Steps.	34
Passing options to the ld command	34
Dynamic and static linking	34
Avoiding naming conflicts during linking	34
Running XL Fortran programs	35
Canceling execution	35
Compiling and executing on different systems.	35
Runtime libraries for POSIX pthreads support.	36
Selecting the language for runtime messages	36
Setting runtime options	36
Other environment variables that affect runtime behavior	47
XL Fortran runtime exceptions	47

Chapter 5. Tracking and reporting compiler usage 49

Understanding utilization tracking and reporting.	49
Overview	49
Four usage scenarios	50
Preparing to use this feature.	58
Time synchronization	58
License types and user information	58
Central configuration	59
Concurrent user considerations.	59
Usage file considerations	60
Regular utilization checking	62
Testing utilization tracking	62
Configuring utilization tracking	63
Editing utilization tracking configuration file entries	64
Understanding the utilization reporting tool	67
Utilization reporting tool command-line options	68
Generating usage reports	71
Understanding usage reports	72
Pruning usage files	74

Diagnostic messages from utilization tracking and reporting 75

Chapter 6. Summary of compiler options by functional category 77

Output control 77
 Input control 78
 Language element control 79
 Floating-point and integer control 81
 Object code control 82
 Error checking and debugging 83
 Listings, messages, and compiler information 85
 Optimization and tuning 86
 Linking 90
 Portability and migration 90
 Compiler customization 91
 Deprecated options 92

Chapter 7. Detailed descriptions of the XL Fortran compiler options 95

-.# 96
 -1 97
 -B 97
 -C 98
 -c 99
 -D 100
 -d 100
 -e 101
 -F 102
 -g 103
 -I 104
 -k 105
 -L 105
 -l 106
 -NS 107
 -O 108
 -o 110
 -p 111
 -q32 112
 -q64 113
 -qalias 114
 -qalias_size 117
 -qalign 118
 -qarch 120
 -qassert 124
 -qattr 125
 -qautodbl 126
 -qbindcextname 128
 -qcache 129
 -qcclines 132
 -qcheck 132
 -qci 133
 -qcompact 134
 -qcr 134
 -qctyplss 135
 -qdbg 136
 -qddim 137
 -qdescriptor 138
 -qdirective 139
 -qdirectstorage 141

-qdlines 142
 -qdpc 142
 -qenum 143
 -qescape 144
 -qessl 146
 -qextern 147
 -qextname 148
 -qfdpr 149
 -qfixed 150
 -qflag 151
 -qfloat 152
 -qfpp 157
 -qflttrap 158
 -qfree 159
 -qfullpath 161
 -qfunctrace 162
 -qfunctrace_xlf_catch 163
 -qfunctrace_xlf_enter 164
 -qfunctrace_xlf_exit 165
 -qhalt 166
 -qhot 167
 -qieee 170
 -qinfo 171
 -qinit 172
 -qinitauto 172
 -qinlgue 175
 -qinline 176
 -qintlog 178
 -qintsize 179
 -qipa 181
 -qkeepparm 187
 -qlanglvl 187
 -qlibansi 189
 -qlibmpi 190
 -qlinedebug 191
 -qlist 192
 -qlistfmt 193
 -qlistopt 195
 -qlog4 196
 -qmaxmem 197
 -qmbcs 198
 -qminimaltoc 199
 -qmixed 200
 -qmkshrobj 200
 -qmoddir 201
 -qmodule 202
 -qnoprint 203
 -qnullterm 204
 -qobject 205
 -qonetrip 206
 -qoptdebug 206
 -qoptimize 207
 -qpdf1, -qpdf2 208
 -qphsinfo 213
 -qpics 215
 -qport 216
 -qposition 218
 -qppsuborigarg 219
 -qprefetch 220
 -qqcount 222
 -qrealsize 223

-qrecur	225
-qreport	226
-qsaa	228
-qsave	229
-qsaveopt.	230
-qsclock	232
-qshowpdf	233
-qsigtrap	234
-qsimd	234
-qsmallstack	236
-qsimp	237
-qsource	242
-qspillsize	243
-qstackprotect	244
-qstacktemp	245
-qstaticlink	246
-qstrict	247
-qstrictieemod	251
-qstrict_induction	252
-qsuffix	253
-qsuppress	254
-qswapomp	256
-qtbtable	257
-qthreaded	258
-qtimestamps	259
-qtune	260
-qundef	262
-qunroll	262
-qunwind.	264
-qversion	264
-qwarn64	266
-qxflag=dvz	266
-qxflag=oldtab	268
-qxf77.	268
-qxf90.	270
-qxf2003	272
-qxlines	275
-qxref	277
-qzerosize	278
-r	279
-S	279
-t	280
-U	281
-u	282
-v	283
-V	283
-W	284
-w	286
-y	286

Chapter 8. Using XL Fortran in a 64-bit environment	289
Compiler options for the 64-bit environment	289

Chapter 9. Problem determination and debugging. 291

Understanding XL Fortran error messages.	291
Error severity	291
Compiler return codes	292
Runtime return codes	292
Format of XL Fortran diagnostic messages.	292
Limiting the number of compile-time messages	293
Selecting the language for messages.	293
Fixing installation or system environment problems	294
Fixing compile-time problems	296
Fixing link-time problems	297
Fixing runtime problems	297
Debugging a Fortran program.	299

Chapter 10. Understanding XL Fortran compiler listings 301

Header section	301
Options section.	301
Source section	302
Error messages	302
PDF report section.	302
Transformation report section	303
Data reorganization report section	305
Attribute and cross reference section	305
Object section	306
File table section	306
Compilation unit epilogue Section	306
Compilation epilogue Section	307

Chapter 11. XL Fortran technical information 309

External names in XL Fortran libraries	309
The XL Fortran runtime environment	309
External names in the runtime environment	309
Technical details of the -qfloat=hsflt option	310
Implementation details for -qautodbl promotion and padding	310
Terminology.	310
Examples of storage relationships for -qautodbl suboptions	311

XL Fortran internal limits 317

Glossary 319

Notices 337

Trademarks and service marks	339
--	-----

Index 341

About this information

This document describes the IBM® XL Fortran for Linux®, V13.1 compiler and explains how to set up the compilation environment and how to compile, link, and run programs written in the Fortran language. This guide also contains cross-references to relevant topics of other reference guides in the XL Fortran documentation suite.

Who should read this information

This information is for anyone who wants to work with the IBM XL Fortran for Linux, V13.1 compiler, is familiar with the Linux operating system, and who has some previous Fortran programming experience. Users new to XL Fortran can also find information on the capabilities and features unique to XL Fortran. This information can help you understand what the features of the compiler are, especially the options, and how to use them for effective software development.

How to use this information

While this information covers topics about configuring the compiler, and compiling, linking and running XL Fortran programs, it does not include information on the following topics, which are covered elsewhere:

- Installation, system requirements, last-minute updates: see the *XL Fortran Installation Guide* and product README.
- Overview of XL Fortran features: see the *Getting Started with XL Fortran*.
- Syntax, semantics, and implementation of the XL Fortran programming language: see the *XL Fortran Language Reference*.
- Optimizing, porting, OpenMP/ SMP programming: see the *XL Fortran Optimization and Programming Guide*.
- Operating system commands related to the use of the compiler: consult your Linux-specific distribution's man page help and documentation.

How this information is organized

This information starts with an overview of the compiler and then outlines the tasks you need to do before invoking the compiler. It then continues with reference information about the compiler options and debugging problems.

This reference includes the following topics:

- Chapter 1, “Introduction,” on page 1 through Chapter 4, “Editing, compiling, linking, and running XL Fortran programs,” on page 21 discuss setting up the compilation environment and the environment variables that you need for different compilation modes, customizing the configuration file, the types of input and output files, compiler listings and messages and information specific to invoking the preprocessor and linkage editor.
- Chapter 6, “Summary of compiler options by functional category,” on page 77 organizes the compiler options by their functional category. You can search for options by their name, or alternatively use the links in the functional category tables and look up options according to their functionality. Chapter 7, “Detailed

descriptions of the XL Fortran compiler options,” on page 95 includes individual descriptions of the compiler options sorted alphabetically. Descriptions provide examples and list related topics.

- Chapter 8, “Using XL Fortran in a 64-bit environment,” on page 289 discusses application development for the 64-bit environment.
- Chapter 9, “Problem determination and debugging,” on page 291 addresses debugging and understanding compiler listings.
- Chapter 11, “XL Fortran technical information,” on page 309 and “XL Fortran internal limits” on page 317 provide information that advanced programmers may need to diagnose unusual problems and run the compiler in a specialized environment.

Conventions

Typographical conventions

The following table explains the typographical conventions used in the IBM XL Fortran for Linux, V13.1 information.







Table 1. Typographical conventions

Typeface	Indicates	Example
bold	Lowercase commands, executable names, compiler options, and directives.	The compiler provides basic invocation commands, <code>xlf</code> , along with several other compiler invocation commands to support various Fortran language levels and compilation environments.
<i>italics</i>	Parameters or variables whose actual names or values are to be supplied by the user. Italics are also used to introduce new terms.	Make sure that you update the <i>size</i> parameter if you return more than the <i>size</i> requested.
<u>underlining</u>	The default setting of a parameter of a compiler option or directive.	<code>nomaf <u>maf</u></code>
monospace	Programming keywords and library functions, compiler builtins, examples of program code, command strings, or user-defined names.	To compile and optimize <code>myprogram.f</code> , enter: <code>xlf myprogram.f -03</code> .
UPPERCASE bold	Fortran programming keywords, statements, directives, and intrinsic procedures. Uppercase letters may also be used to indicate the minimum number of characters required to invoke a compiler option/suboption.	The ASSERT directive applies only to the DO loop immediately following the directive, and not to any nested DO loops.

Qualifying elements (icons and bracket separators)

In descriptions of language elements, this information uses marked bracket separators to delineate large blocks of text and icons to delineate small segments of text as follows:

Table 2. Qualifying elements

Bracket separator text	Icon	Meaning
Fortran 2003 begins / ends	 	The text describes an IBM XL Fortran implementation of the Fortran 2003 standard.
Fortran 95 begins / ends	 	The text describes an IBM XL Fortran implementation of the Fortran 95 standard.
IBM extension begins / ends	 	The text describes a feature that is an IBM XL Fortran extension to the standard language specifications.


Syntax diagrams


Throughout this information, diagrams illustrate XL Fortran syntax. This section will help you to interpret and use those diagrams.



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

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

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

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

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

Fragments, which are diagrams of syntactical units other than complete commands, directives, or statements, start with the  symbol and end with the  symbol.

IBM XL Fortran extensions are marked by a number in the syntax diagram with an explanatory note immediately following the diagram.

Program units, procedures, constructs, interface blocks and derived-type definitions consist of several individual statements. For such items, a box encloses the syntax representation, and individual syntax diagrams show the required order for the equivalent Fortran statements.

- Required items are shown on the horizontal line (the main path):



- Optional items are shown below the main path:



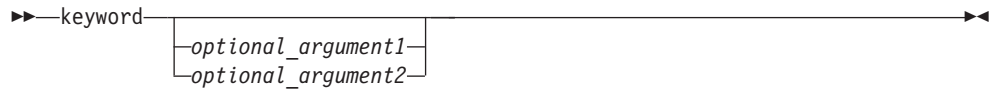
Note: Optional items (not in syntax diagrams) are enclosed by square brackets ([and]). For example, [UNIT=]u

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

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



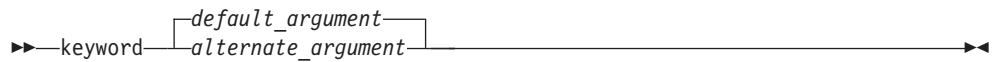
If choosing one of the items is optional, the entire stack is shown below the main path.



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



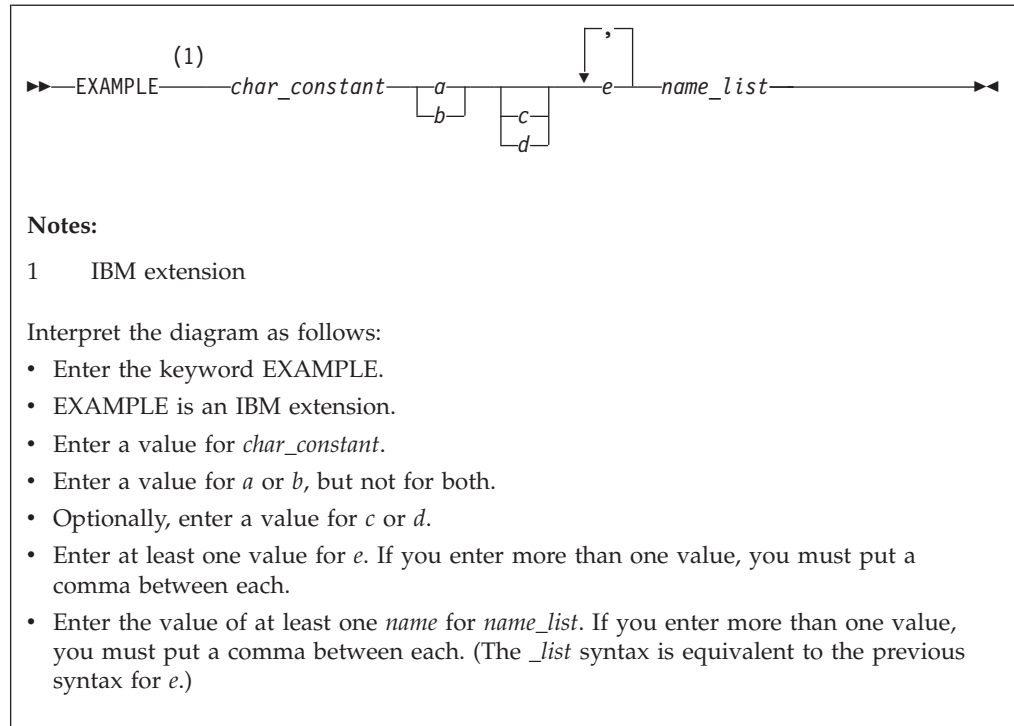
- The item that is the default is shown above the main path.



- Keywords are shown in nonitalic letters and should be entered exactly as shown.
- Variables are shown in italicized lowercase letters. They represent user-supplied names or values. If a variable or user-specified name ends in *_list*, you can provide a list of these terms separated by commas.
- If punctuation marks, parentheses, arithmetic operators, or other such symbols are shown, you must enter them as part of the syntax.

Sample syntax diagram

The following is an example of a syntax diagram with an interpretation:



How to read 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.

Examples in this information

The examples in this information, except where otherwise noted, are coded in a simple style that does not try to conserve storage, check for errors, achieve fast performance, or demonstrate all possible methods to achieve a specific result.

The examples for installation information are labelled as either *Example* or *Basic example*. *Basic examples* are intended to document a procedure as it would be performed during a basic, or default, installation; these need little or no modification.

Notes on the terminology used

Some of the terminology in this information is shortened as follows:

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

Related information

The following sections provide related information for XL Fortran:

IBM XL Fortran information

XL Fortran provides product information in the following formats:

- README files

README files contain late-breaking information, including changes and corrections to the product information. README files are located by default in the XL Fortran directory and in the root directory of the installation CD.

- Installable man pages

Man pages are provided for the compiler invocations and all command-line utilities provided with the product. Instructions for installing and accessing the man pages are provided in the *IBM XL Fortran for Linux, V13.1 Installation Guide*.

- Information center

The information center of searchable HTML files can be launched on a network and accessed remotely or locally. Instructions for installing and accessing the online information center are provided in the *IBM XL Fortran for Linux, V13.1 Installation Guide*.

The information center is viewable on the Web at <http://publib.boulder.ibm.com/infocenter/lxpcmp/v111v131/index.jsp>.

- PDF documents

PDF documents are located by default in the `/opt/ibmcmp/xf/13.1/doc/LANG/pdf/` directory, where *LANG* is one of `en_US` or `ja_JP`. The PDF files are also available on the Web at <http://www.ibm.com/software/awdtools/fortran/xlfortran/linux/library/>.

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

Table 3. XL Fortran PDF files

Document title	PDF file name	Description
<i>IBM XL Fortran for Linux, V13.1 Installation Guide, GI11-7916-00</i>	install.pdf	Contains information for installing XL Fortran and configuring your environment for basic compilation and program execution.
<i>Getting Started with IBM XL Fortran for Linux, V13.1, GI11-7915-00</i>	getstart.pdf	Contains an introduction to the XL Fortran product, with information on setting up and configuring your environment, compiling and linking programs, and troubleshooting compilation errors.
<i>IBM XL Fortran for Linux, V13.1 Compiler Reference, SC23-8609-00</i>	compiler.pdf	Contains information about the various compiler options and environment variables.
<i>IBM XL Fortran for Linux, V13.1 Language Reference, SC23-8610-00</i>	langref.pdf	Contains information about the Fortran programming language as supported by IBM, including language extensions for portability and conformance to nonproprietary standards, compiler directives and intrinsic procedures.
<i>IBM XL Fortran for Linux, V13.1 Optimization and Programming Guide, SC23-8611-00</i>	proguide.pdf	Contains information on advanced programming topics, such as application porting, interlanguage calls, floating-point operations, input/output, application optimization and parallelization, and the XL Fortran high-performance libraries.

To read a PDF file, use the Adobe® Reader. If you do not have the Adobe Reader, you can download it (subject to license terms) from the Adobe Web site at <http://www.adobe.com>.

More information related to XL Fortran including IBM Redbooks® publications, white papers, tutorials, and other articles, is available on the Web at:

<http://www.ibm.com/software/awdtools/fortran/xlfortran/linux/library/>

Standards and specifications

XL Fortran is designed to support the following standards and specifications. You can refer to these standards for precise definitions of some of the features found in this information.

- *American National Standard Programming Language FORTRAN, ANSI X3.9-1978.*
- *American National Standard Programming Language Fortran 90, ANSI X3.198-1992.*
- *ANSI/IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-1985.*
- *Federal (USA) Information Processing Standards Publication Fortran, FIPS PUB 69-1.*
- *Information technology - Programming languages - Fortran, ISO/IEC 1539-1:1991 (E). (This information uses its informal name, Fortran 90.)*
- *Information technology - Programming languages - Fortran - Part 1: Base language, ISO/IEC 1539-1:1997. (This information uses its informal name, Fortran 95.)*
- *Information technology - Programming languages - Fortran - Part 1: Base language, ISO/IEC 1539-1:2004. (This information uses its informal name, Fortran 2003.)*
- *Military Standard Fortran DOD Supplement to ANSI X3.9-1978, MIL-STD-1753 (United States of America, Department of Defense standard). Note that XL*

Fortran supports only those extensions documented in this standard that have also been subsequently incorporated into the Fortran 90 standard.

- *OpenMP Application Program Interface Version 3.0*, available at <http://www.openmp.org>

Other IBM information

- *ESSL for AIX V4.4 - ESSL for Linux on POWER V4.4 Guide and Reference* available at the Engineering and Scientific Subroutine Library (ESSL) and Parallel ESSL Web page.

Technical support

Additional technical support is available from the XL Fortran Support page at <http://www.ibm.com/software/awdtools/fortran/xlfortran/linux/support/>. This page provides a portal with search capabilities to a large selection of Technotes and other support information.

If you cannot find what you need, you can send e-mail to compinfo@ca.ibm.com.

For the latest information about XL Fortran, visit the product information site at <http://www.ibm.com/software/awdtools/fortran/xlfortran/linux/>.

How to send your comments

Your feedback is important in helping to provide accurate and high-quality information. If you have any comments about this information or any other XL Fortran information, send your comments by e-mail to compinfo@ca.ibm.com.

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

Chapter 1. Introduction

IBM XL Fortran for Linux, V13.1 is an optimizing, standards-based, command-line compiler for the Linux operating system, running on Power hardware with the Power architecture. The XL Fortran compiler enables application developers to create and maintain optimized 32-bit and 64-bit applications for the Linux operating system. The compiler also offers a diversified portfolio of optimization techniques that allow an application developer to exploit the multi-layered architecture of the Power processor.

The implementation of the Fortran programming language is intended to promote portability among different environments by enforcing conformance to language standards. A program that conforms strictly to its language specification has maximum portability among different environments. In theory, a program that compiles correctly with one standards-conforming compiler will compile and execute correctly under all other conforming compilers, insofar as hardware differences permit. A program that correctly exploits the extensions to the programming language in which it is written can improve the efficiency of its object code.

XL Fortran can be used for large, complex, and computationally intensive programs. It also supports interlanguage calls with C/C++. For applications that require SIMD (single-instruction, multiple data) parallel processing, performance improvements can be achieved through optimization techniques, which may be less labor-intensive than vector programming. Many of the optimizations developed by IBM are controlled by compiler options and directives.

Chapter 2. 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 V13.1 compiler is supported on several Linux distributions. See the *XL Fortran for Linux Installation Guide* and README file for a list of supported distributions and requirements.

The compiler, its generated object programs, and runtime library can run on all POWER3™, POWER4™, POWER5™, POWER5+™, POWER6®, POWER7™, PowerPC® 970, and PowerPC systems with the required software, disk space, and virtual storage.

The POWER3, POWER4, POWER5, POWER5+, POWER6, and POWER7 processors are a type of PowerPC processor. In this document, any statement or reference to the PowerPC processor also applies to the POWER3, POWER4, POWER5, POWER5+, POWER6, or POWER7 processor.

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

Language support

The XL Fortran language consists of the following:

- The full American National Standard Fortran 90 language (referred to as Fortran 90 or F90), defined in the documents *American National Standard Programming Language Fortran 90, ANSI X3.198-1992* and *Information technology - Programming languages - Fortran, ISO/IEC 1539-1:1991 (E)*. This language has a superset of the features found in the FORTRAN 77 standard. It adds many more features that are intended to shift more of the tasks of error checking, array processing, memory allocation, and so on from the programmer to the compiler.
- The full ISO Fortran 95 language standard (referred to as Fortran 95 or F95), defined in the document *Information technology - Programming languages - Fortran - Part 1: Base language, ISO/IEC 1539-1:1997*.
- The full ISO Fortran 2003 language standard (referred to as Fortran 2003 or F2003), defined in the document *Information technology - Programming languages - Part 1: Base language, ISO/IEC 1539-1:2004*.
- Extensions to the Fortran standard:
 - Common Fortran language extensions defined by other compiler vendors, in addition to those defined by IBM
 - Industry extensions that are found in Fortran products from various compiler vendors
 - Extensions specified in SAA Fortran

In the *XL Fortran Language Reference*, extensions to the Fortran 95 language and Fortran 2003 language are marked as described in the *Conventions: Qualifying elements* section.

Migration support

The XL Fortran compiler helps you to port or to migrate source code among Fortran compilers by providing full Fortran 90, Fortran 95, and Fortran 2003 language support, and selected language extensions (including intrinsic functions and data types) from many different compiler vendors. Throughout this document, we 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 `f77`, `fort77`, `xlF`, and `xlF_r` commands provide maximum compatibility with existing FORTRAN 77 programs. The default options provided with the `f90`, `xlF90`, and `xlF90_r` commands give access to the full range of Fortran 90 language features. The default options provided with the `f95`, `xlF95`, and `xlF95_r` commands give access to the full range of Fortran 95 language features. The default options provided with the `f2003`, `xlF2003`, and `xlF2003_r` commands give access to the full range of Fortran 2003 language features.

Additionally, you can name your source files with extensions such as `.f77`, `.f90`, `.f95`, or `.f03` and use the generic compiler invocations such as `xlF` or `xlF_r` to automatically select language-level appropriate defaults.

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, Fortran 95, or Fortran 2003 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), full Fortran 95 standard (`-qlanglvl=95std` option), and full Fortran 2003 Standard (`-qlanglvl=2003std` option)
- Fortran 90, less any obsolescent features (`-qlanglvl=90pure` option)
- Fortran 95, less any obsolescent features (`-qlanglvl=95pure` option)
- Fortran 2003, less any obsolescent features (`-qlanglvl=2003pure` option)
- IBM SAA FORTRAN (`-qsaa` option)

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

Note: Fortran 2003 conformance checking is based on XL Fortran's current subset implementation of this standard.

Highly configurable compiler

You can invoke the compiler by using the following commands:

- `xlF`
- `xlF_r`
- `xlF90`

- `xlF90_r`
- `f90`
- `xlF95`
- `xlF95_r`
- `f95`
- `xlF2003`
- `xlF2003_r`
- `f2003`
- `f77`
- `fort77`

The `xlF`, `xlF_r`, `f77`, and `fort77` commands maintain maximum compatibility with the behavior and I/O formats of XL Fortran Version 2. The `xlF90`, `xlF90_r`, and `f90` commands provide more Fortran 90 conformance and some implementation choices for efficiency and usability. The `f95`, `xlF95` and `xlF95_r` commands provide more Fortran 95 conformance and some implementation choices for efficiency and usability. The `xlF2003`, `xlF2003_r`, and `f2003` commands provide more Fortran 2003 conformance and some implementation choices for efficiency and usability. The `f77` or `fort77` command provides maximum compatibility with the XPG4 behavior.

The main difference between the set of `xlF_r`, `xlF90_r`, `xlF95_r`, and `xlF2003_r` commands and the set of `xlF`, `xlF90`, `f90`, `xlF95`, `f95`, `xlF2003`, `f2003`, `f77`, and `fort77` commands is that the first set links and binds the object files to the threadsafe 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/13.1/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 edit the default configuration file or use a customized configuration file instead of creating many separate aliases or shell scripts.

Related information:

- “Using custom compiler configuration files” on page 10
- “Compiling XL Fortran programs” on page 21
- Chapter 6, “Summary of compiler options by functional category,” on page 77 and Chapter 7, “Detailed descriptions of the XL Fortran compiler options,” on page 95

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 “Listings, messages, and compiler information” on page 85 and Chapter 10, “Understanding XL Fortran compiler listings,” on page 301.

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

Symbolic debugger support

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

Program optimization

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

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

The *XL Fortran Optimization and Programming Guide* provides a road map and optimization strategies for these features.

Chapter 3. Setting up and customizing XL Fortran

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

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

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

Where to find installation instructions

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

1. Read the file called `/opt/ibmcmp/xlf/13.1/doc/en_US/README`, and follow any directions it gives. It contains information that you must know and possibly distribute to other people who use XL Fortran.
2. Read the *XL Fortran Installation Guide* to see if there are any important notices you need to be aware of or any updates you might need to apply to your system before doing the installation.
3. You should be familiar with the RPM Package Manager (RPM) for installing this product. For information on using RPM, visit the RPM Web page at URL <http://www.rpm.org/>, or type `rpm --help` at the command line.

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

Using the compiler on a network file System

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

The following directories contain XL Fortran components:

- `/opt/ibmcmp/xlf/13.1/bin` contains the compiler invocation commands.
- `/opt/ibmcmp/xlf/13.1/exe` contains executables and files that the compiler needs.
- `/opt/ibmcmp/xlf/13.1/lib` and `/opt/ibmcmp/xlf/13.1/lib64` contain the non-redistributable libraries.
- `/opt/ibmcmp/lib/` and `/opt/ibmcmp/lib64/` contain the redistributable libraries.
- `/opt/ibmcmp/xlf/13.1/include` contains the include files and supplied `.mod` files.
- `/opt/ibmcmp/msg` contains the message catalogues for the redistributable runtime libraries.

You must copy the `/opt/ibmcmp/xlf/13.1/etc/xlf.cfg` file from the server to the client. The `/opt/ibmcmp/xlf/13.1/etc/` directory contains the configuration files specific to a machine, and it cannot be mounted from the server.

Correct settings for environment variables

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

Environment variable basics

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

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

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

Environment variables for national language support

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

LANG

Specifies the *locale*. A locale is divided into categories. Each category contains a specific aspect of the locale data. Setting **LANG** might change the national language for all the categories.

NLSPATH

Refers to a list of directory names where the message catalogs might be found.

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

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

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

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

For example:

LC_MESSAGES

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

LC_TIME

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

LC_CTYPE

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

LC_NUMERIC

Specifies the format to use for input and output of numeric values.

Note:

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

Related information: “Selecting the language for runtime messages” on page 36.

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

Setting library search paths

If your executable program is linked with shared libraries, you must set the runtime library search paths. You can use one of the following three ways to set runtime library search paths:

- When linking the shared library into the executable, use the **-R** (or **-rpath**) compiler/link option.
- Before linking the shared library into the executable, set the **LD_RUN_PATH** environment variable.
- Set the **LD_LIBRARY_PATH** environment variable.

For example:

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

```
# -L directories are searched at link time for both static and shared libraries.
# -R directories are searched at run time for shared libraries.
```

For more information about the linker option **-R** (or **-rpath**), and environment variables **LD_RUN_PATH** and **LD_LIBRARY_PATH**, see the man pages for the **ld** command.

PDF environment variables

The following list includes PDF environment variables that you can use with the **-qpdf** compiler option:

- **PDF_BIND_PROCESSOR:**

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

- **PDFDIR:**

When you compile a Fortran 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.

Problems can occur if the profiling information is stored in the wrong place or is updated by more than one application. To avoid these problems, you must follow these guidelines:

- Always set the **PDFDIR** environment variable when using the **-qpdf** option.
- Store the profiling information for each application in a different directory, or use the **-qpdf1=pdfname**, **-qpdf2=pdfname** option to explicitly name the temporary profiling files according to the template provided.
- Leave the value of the **PDFDIR** environment variable unchanged 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 runtime 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* in the *XL Fortran Optimization and Programming Guide* 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 runtime option **unit_vars=yes**; then set one or more environment variables with names of the form **XLFUNIT_unit** to file names. See *Naming files that are connected with no explicit name* in the *XL Fortran Optimization and Programming Guide* for examples.

Using custom compiler configuration files

The XL Fortran compiler generates a default configuration file `/opt/ibmcmp/xf/13.1/etc/xf.cfg.$OSRelease.gcc$gccVersion`. For example, `/opt/ibmcmp/xf/13.1/etc/xf.cfg.sles11.gcc432` or `/opt/ibmcmp/xf/13.1/etc/xf.cfg.rhel5.5.gcc412` at installation time. (See the *XL Fortran Installation Guide* for more information on the various tools you can use to generate the configuration file during installation.) The configuration file specifies information that the compiler uses when you invoke it.

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

If you want users to be able to choose among several sets of compiler options, you might want to use custom configuration files for specific needs. For example, you might want to enable **-qlist** by default for compilations using the **xlF** compiler invocation command. This is to avoid forcing your users to specify this option on the command line for every compilation. Because **-qlist** is automatically in effect every time the compiler is called with the **xlF** command.

You have several options for customizing configuration files:

- You can directly edit the default configuration file. In this case, the customized options will apply for all users for all compilations. The disadvantage of this option is that you will need to reapply your customizations to the new default configuration file that is provided every time you install a compiler update.
- You can use the default configuration file as the basis of customized copies that you specify at compile time with the **-F** option. In this case, the custom file overrides the default file on a per-compilation basis.

Note: This option requires you to reapply your customization after you apply service to the compiler.

- You can create custom, or user-defined, configuration files that are specified at compile time with the **XL_FUSR_CONFIG** environment variable. In this case, the custom user-defined files complement, rather than override, the default configuration file, and they can be specified on a per-compilation or global basis. The advantage of this option is that you do not need to modify your existing, custom configuration files when a new system configuration file is installed during an update installation. Procedures for creating custom, user-defined configuration files are provided below.

Related information:

- “-F” on page 102

Creating custom configuration files

If you use the **XL_FUSR_CONFIG** environment variable to instruct the compiler to use a custom user-defined configuration file, the compiler examines and processes the settings in that user-defined configuration file before looking at the settings in the default system configuration file.

To create a custom user-defined configuration file, you add stanzas which specify multiple levels of the **use** attribute. The user-defined configuration file can reference definitions specified elsewhere in the same file, as well as those specified in the system configuration file. For a given compilation, when the compiler looks for a given stanza, it searches from the beginning of the user-defined configuration file and follows any other stanza named in the **use** attribute, including those specified in the system configuration file.

If the stanza named in the **use** attribute has a name different from the stanza currently being processed, the search for the **use** stanza starts from the beginning of the user-defined configuration file. This is the case for stanzas A, C, and D which you see in the following example. However, if the stanza in the **use** attribute has the same name as the stanza currently being processed, as is the case of the two B stanzas in the example, the search for the **use** stanza starts from the location of the current stanza.

The following example shows how you can use multiple levels for the **use** attribute. This example uses the **options** attribute to help show how the **use** attribute works, but any other attributes, such as **libraries** can also be used.

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

Figure 1. Sample configuration file

In this example:

- stanza A uses option sets *A* and *Z*
- stanza B uses option sets *B1*, *B2*, *D*, *A*, and *Z*
- stanza C uses option sets *C*, *A*, and *Z*
- stanza D uses option sets *D*, *A*, and *Z*

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

By default, values defined in a stanza in a configuration file are added to the list of values specified in previously processed stanzas. For example, assume that the `XLF_USR_CONFIG` environment variable is set to point to the user-defined configuration file at `~/userconfig1`. With the user-defined and default configuration files shown in the following example, the compiler references the `xlf` stanza in the user-defined configuration file and uses the option sets specified in the configuration files in the following order: *A1*, *A*, *D*, and *C*.

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

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

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

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

DEFLT:
     options=<C>
```

Figure 3. Default configuration file `xlf.cfg`

Overriding the default order of attribute values

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

Table 4. Assignment operators and attribute ordering

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

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

Custom user-defined configuration file

~/userconfig2

Default configuration file xlf.cfg

```
xlf_prepend: use=xlf
             options--<B1>
xlf_replace: use=xlf
             options:=<B2>
xlf_append:  use=xlf
             options+=<B3>
```

```
xlf: use=DEFLT
     options=<B>
DEFLT:
     options=<C>
```

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

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

1. stanza xlf uses B, D, and C
2. stanza xlf_prepend uses B1, B, D, and C
3. stanza xlf_replace uses B2
4. stanza xlf_append uses B, D, C, and B3

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

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

Figure 4. Using additional assignment operations

Examples of stanzas in custom configuration files

```
DEFLT: use=DEFLT
       options = -g
```

This example specifies that the **-g** option is to be used in all compilations.

```
xlf: use=xlf
     options+=-qlist
xlf_r: use=xlf_r
     options+=-qlist
```

This example specifies that **-qlist** is to be used for any compilation called by the **xlf** and **xlf_r** commands. This **-qlist** specification overrides the default setting of **-qlist** specified in the system configuration file.

```
DEFLT: use=DEFLT
       libraries=-L/home/user/lib,-lmylib
```

This example specifies that all compilations should link with /home/user/lib/libmylib.a.

Editing the default configuration file

The configuration file specifies information that the compiler uses when you invoke it. XL Fortran provides the default configuration file `/opt/ibmcomp/xlf/13.1/etc/xlf.cfg` at installation time.

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

```
ln -s /opt/ibmcomp/xlf/13.1/bin/xlf95 /home/username/bin/xlf95
```

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

Note:

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

Attributes

The configuration file contains the following attributes:

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

gcc_libs_64

When invoked in 64-bit mode, the linker options to specify the path to the GCC libraries and to link the GCC library.

gcc_path

Specifies the path to the 32-bit tool chain.

gcc_path_64

Specifies the path to the 64-bit tool chain.

cpp The absolute path name of the C preprocessor, which is automatically called for files ending with a specific suffix (usually **.F**).

xlf The absolute path name of the main compiler executable file. The compiler commands are driver programs that execute this file.

code The absolute path name of the optimizing code generator.

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

as The absolute path name of the assembler.

asopt Lists names of options that are assumed to be assembler options for cases where, for example, a compiler option and an assembler option use the same letter. The list is a concatenated set of single-letter flags. Any flag that takes an argument is followed by a colon, and the whole list is enclosed by double quotation marks. You may find it more convenient to set up this attribute than to pass options to the assembler through the **-W** compiler option.

ld The absolute path name of the linker.

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

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

options

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

cppoptions

A string of options that are separated by commas, to be processed by **cpp** (the C preprocessor) as if you entered them on the command line before any other option. This attribute is needed because some **cpp** options are usually required to produce output that can be compiled by XL Fortran. The default is **-C**, which preserves any C-style comments in the output. Also, refer to the “**-qfpp**” on page 157 and “**-qppsuborigarg**” on page 219 options for other useful **cpp** options.

Note: You can specify **-C!** preprocessor option on the command line (**-WF, -C!**) to override the default setting.

fsuffix The allowed suffix for Fortran source files. The default is **f**. The compiler requires that all source files in a single compilation have the same suffix. Therefore, to compile files with other suffixes, such as **f95**, you must change this attribute in the configuration file or use the **-qsuffix** compiler option. For more information on **-qsuffix**, see “-qsuffix” on page 253.

cppsuffix The suffix that indicates a file must be preprocessed by the C preprocessor (**cpp**) before being compiled by XL Fortran. The default is **F**.

osuffix The suffix used to recognize object files that are specified as input files. The default is **o**.

ssuffix The suffix used to recognize assembler files that are specified as input files. The default is **s**.

libraries **-l** options, which are separated by commas, that specify the libraries used to link all programs.

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

hot Absolute path name of the program that does array language transformations.

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

bolt Absolute path name of the binder.

defaultmsg Absolute path name of the default message files.

include Indicates the search path that is used for compilation include files and module files.

include_32 Indicates the search path that is used for 32-bit compilation include files.

include_64 Indicates the search path that is used for 64-bit compilation include files.

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

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

Related information: You can use the “-F” on page 102 option to select a different configuration file, a specific stanza in the configuration file, or both.

Determining which level of XL Fortran is installed

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

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

```
rpm -qa | grep xlf.cmp-13.1 | xargs rpm -qi
```

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

You can also use the **-qversion** compiler option to display the version, release, and level of the compiler and its components.

Backward compatibility issues

This section describes backward compatibility issues and their workarounds.

Compiler option compatibility issues

In IBM XL Fortran for Linux, V13.1, the implementation of the threadprivate data, that is, OpenMP threadprivate variable, has been improved. The operating system thread local storage is used instead of the runtime implementation. The new implementation might improve performance on some applications.

If you plan to mix the object files `.o` that you have compiled with levels prior to 13.1 with the object files that you compiled with IBM XL Fortran for Linux, V13.1, and the same OpenMP threadprivate variables are referenced in both old and new object files, different implementations might cause incompatibility issues. A link error, a compile time error or other undefined behaviors might occur. To support backward compatibility, you can use the **-qsmp=noostls** suboption to switch back to the old implementation. You can recompile the entire program with the default suboption **-qsmp=ostls** to get the benefit of the new implementation.

If you are not sure whether the object files you have compiled with levels prior to IBM XL Fortran for Linux, V13.1 contain any old implementation, you can use the **readelf -s** command to determine whether you need to use the **-qsmp=noostls** suboption. The following code is an example that shows how to use the **readelf -s** command:

```
> readelf -s oldfiles.o
...
._xlGetThStorageBlock U          -
._xlGetThValue         U          -
...
```

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

Binary compatibility issues with previous releases

Because of a change to the signature of a compiler-generated routine, mixing code compiled with XL Fortran V11.1 or V12.1 with code containing parameterized derived types compiled with the latest compiler may require recompiling all the source files. Otherwise, a runtime error will occur if the older code uses certain polymorphic references that can resolve to parameterized derived type objects where a length parameter is involved.

For example, the new application extends a derived type whose declaration was compiled using either XL Fortran V11.1 or XL Fortran V12.1, and the extending

type has length derived type parameters, and the new application passes an object that has a dynamic type of the extending type through polymorphism via argument association or function reference to a procedure compiled using XL Fortran V11.1 or XL Fortran V12.1 compiler version.

XL Fortran runtime detects the above issue and halts the execution with the following error message:

XL Fortran detected a mismatch in a compiler-generated routine. If your code contains compilation units compiled with XL Fortran V11.1 or V12.1, recompile them with the latest XL Fortran compiler.

Example of argument association:

<pre> 11.1/12.1 module m type base integer i end type contains subroutine sub(arg) class(base) :: arg class(base), allocatable :: local allocate(local, source=arg) end subroutine end module </pre>	<pre> 13.1 use m type, extends(base) :: child (1) integer, len :: l integer :: m(1) integer :: n(1) end type class(base), allocatable :: b1 allocate(child(5) :: b1) call sub(b1) end </pre>
---	--

The problem is the ALLOCATE statement in 11.1/12.1 code is calling a compiler-generated routine that is compiled using newer releases of XL Fortran compiler other than XL Fortran V11.1 or XL Fortran V12.1 through type bound procedure call. Since the signature of the compiler-generated routine is different between 11.1/12.1 releases and newer releases, module m needs to be recompiled using newer releases of XL Fortran compiler.

Example of function call:

<pre> 11.1/12.1 module m type base integer i contains procedure :: bar => bar_base end type type container class(base), allocatable :: b1 end type contains subroutine bar_base(a) class(base) a print *, "base" end subroutine end module program main use m interface function foo() import type(container) :: foo end function end interface type(container) :: c1 c1 = foo() call c1%b1%bar end program </pre>	<pre> 13.1 module n use m type, extends(base) :: child(1) integer, len :: l integer a(1) contains procedure :: bar => bar_child end type contains subroutine bar_child(a) class(child(*)) a print *, "child" print *, a%a end subroutine end module function foo() use n type(container) :: foo allocate(child(6) :: foo%b1) end function </pre>
--	--

The problem is `c1 = foo()` in program main is calling a compiler-generated routine that is compiled using newer releases of XL Fortran compiler other than V11.1 or V12.1 through type bound procedure call via function foo. Since the signature of the compiler-generated routine is different between 11.1/12.1 releases and newer releases, program main needs to be recompiled using newer releases of XL Fortran compiler.

Running two levels of XL Fortran

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

To do this, consult the *XL Fortran Installation Guide* for details.

Chapter 4. Editing, compiling, linking, and running XL Fortran programs

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

Prerequisite information:

1. Before you can use the compiler, all the required Linux settings (for example, certain environment variables and storage limits) must be correct for your user ID; for details, see “Correct settings for environment variables” on page 8.
2. To learn more about writing and optimizing XL Fortran programs, refer to the *XL Fortran Language Reference* and *XL Fortran Optimization and Programming Guide*.

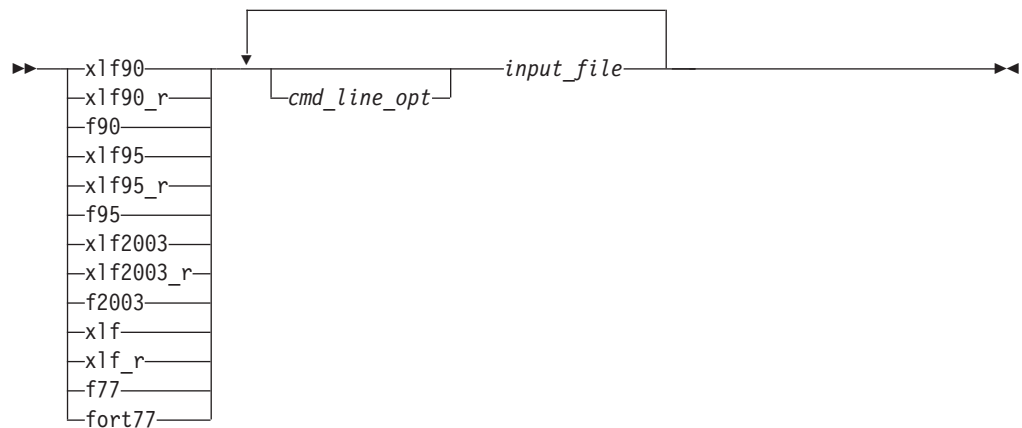
Editing XL Fortran source files

To create Fortran source programs, you can use any of the available text editors, such as **vi** or **emacs**. Source programs must have a suffix of **.f** unless the **fsuffix** attribute in the configuration file specifies a different suffix, or the **-qsuffix** compiler option is used. You can also use a suffix of **.F** if the program contains C preprocessor (**cpp**) directives that must be processed before compilation begins. Source files with the **.f77**, **.f90**, **.f95**, or **.f03** suffix are also valid.

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

Compiling XL Fortran programs

To compile a source program, use one of the **xlf90**, **xlf90_r**, **f90**, **xlf95**, **xlf95_r**, **f95**, **xlf2003**, **xlf2003_r**, **f2003**, **xlf**, **xlf_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 configuration file `/opt/ibmcomp/xlf/13.1/etc/xlf.cfg`).

The invocation command performs the necessary steps to compile the Fortran source files, assemble any `.s` files, and link the object files and libraries into an executable program. In particular, the `xlf_r`, `xlf90_r`, `xlf95_r`, and `xlf2003_r` commands use the components for multi-threading (libraries, and so on) to link and bind object files.

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

Table 5. XL Fortran Invocation commands

Driver Invocation	Path or Location	Chief Functionality	Linked Libraries
<code>xlf90</code> , <code>f90</code>	<code>/opt/ibmcmp/xlf/13.1/bin</code>	Fortran 90	<code>libxlf90.so</code>
<code>xlf90_r</code>	<code>/opt/ibmcmp/xlf/13.1/bin</code>	Threadsafe Fortran 90	<code>libxlf90_r.so</code>
<code>xlf95</code> , <code>f95</code>	<code>/opt/ibmcmp/xlf/13.1/bin</code>	Fortran 95	<code>libxlf90.so</code>
<code>xlf95_r</code>	<code>/opt/ibmcmp/xlf/13.1/bin</code>	Threadsafe Fortran 95	<code>libxlf90_r.so</code>
<code>xlf2003</code>	<code>/opt/ibmcmp/xlf/13.1/bin</code>	Fortran 2003	<code>libxlf90.so</code>
<code>xlf2003_r</code>	<code>/opt/ibmcmp/xlf/13.1/bin</code>	Threadsafe Fortran 2003	<code>libxlf90.so</code>
<code>f2003</code>	<code>/opt/ibmcmp/xlf/13.1/bin</code>	Fortran 2003	<code>libxlf90.so</code>
<code>xlf</code>	<code>/opt/ibmcmp/xlf/13.1/bin</code>	FORTTRAN 77	<code>libxlf90.so</code>
<code>xlf_r</code>	<code>/opt/ibmcmp/xlf/13.1/bin</code>	Threadsafe FORTTRAN 77	<code>libxlf90_r.so</code>
<code>f77</code> , <code>fort77</code>	<code>/opt/ibmcmp/xlf/13.1/bin</code>	FORTTRAN 77	<code>libxlf90.so</code>

The invocation commands have the following implications for directive triggers:

- For `f77`, `fort77`, `f90`, `f95`, `f2003`, `xlf`, `xlf90`, `xlf95`, and `xlf2003`, the directive trigger is **IBM*** by default.
- For all other commands, the directive triggers are **IBM*** and **IBMT** by default. If you specify `-qsmp`, the compiler also recognizes the **IBMP**, **SMP\$**, and **\$OMP** trigger constants. If you specify the `-qsmp=omp` option, the compiler only recognizes the **\$OMP** trigger constant.

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

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

`libxlf90_t.so` is provided for both threaded and non-threaded applications. XL Fortran determines at run time whether your application is threaded.

XL Fortran provides the library `libxlf90_t.so`, in addition to `libxlf90_r.so`. `libxlf90_t.so` exports the same entry points as `libxlf90_r.so` does. The library `libxlf90_r.so` is a superset of `libxlf90_t.so`. The file `xlf.cfg` is set up to link to `libxlf90_r.so` automatically when you use the `xlf90_r`, `xlf95_r`, and `xlf_r` commands. `libxlf90_t.so` is a partial thread-support runtime library. Unlike

`libxlf90_r.so`, `libxlf90_t.so` does not provide thread synchronization and routines in `libxlf90_t.so` are not thread-reentrant. Therefore, only one Fortran thread at a time can perform I/O operations or invoke Fortran intrinsics. You can use `libxlf90_t.so` instead of `libxlf90_r.so` in multithread applications where there is only one Fortran thread, to avoid the thread synchronization overhead in `libxlf90_r.so`.

When you bind a multithreaded executable with multiple Fortran threads, `libxlf90_r.so` should be used. Note that using the `xlf_r`, `xlf90_r`, `xlf95_r`, or `xlf2003_r` invocation command ensures the proper linking.

Compiling Fortran 90 or Fortran 95 programs

The `f90`, `xlf90`, and `xlf90_r` commands make your programs conform more closely to the Fortran 90 standard than do the other invocation commands. The `f95`, `xlf95`, and `xlf95_r` commands make your programs conform more closely to the Fortran 95 standard than do the other invocation commands. `f90`, `xlf90`, `xlf90_r`, `f95`, `xlf95`, and `xlf95_r` are the preferred commands for compiling any new programs. They all accept Fortran 90 free source form by default; to use them for fixed source form, you must use the `-qfixed` option. I/O formats are slightly different between these commands and the other commands. I/O formats also differ between the set of `f90`, `xlf90` and `xlf90_r` commands and the set of `f95`, `xlf95` and `xlf95_r` commands. We recommend that you switch to the Fortran 95 formats for data files whenever possible.

By default, the `f90`, `xlf90`, and `xlf90_r` commands do not conform completely to the Fortran 90 standard. Also, by default, the `f95`, `xlf95`, and `xlf95_r` commands do not conform completely to the Fortran 95 standard. If you need full Fortran 90 or Fortran 95 compliance, compile with any of the following additional compiler options (and suboptions):

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

Also, specify the following runtime options before running the program, with a command similar to one of the following:

```
export XLF RTEOPTS="err_recovery=no:langlvl=90std"
export XLF RTEOPTS="err_recovery=no:langlvl=95std"
```

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.

Compiling Fortran 2003 programs

The `f2003`, `xlf2003`, and `xlf2003_r` commands make your programs conform more closely to the Fortran 2003 Standard than do the other invocation commands. The Fortran 2003 commands accept free source form by default. I/O formats for the Fortran 2003 commands are the same as for the `f95`, `xlf95`, and `xlf95_r` commands. The Fortran 2003 commands format infinity and NaN floating-point values differently from previous commands. The Fortran 2003 commands enable polymorphism by default.

By default, the `f2003`, `xlf2003`, and `xlf2003_r` commands do not conform completely to the Fortran 2003 standard. If you need full compliance, compile with the following additional compiler suboptions:

```
-qlanglvl=2003std -qnodirective -qnoescape -qfloat=nomaf:rndsngl:nofold
-qnoswapomp -qstrictieemod
```

Also specify the following runtime options:

```
XLFRTEOPTS="err_recovery=no:langlvl=2003std:iostat_end=2003std:
internal_nldelim=2003std"
```

Compiling and linking a library

Compiling a static library

To compile a static library:

1. Compile each source file into an object file, with no linking. For example:

```
xlf -c bar.f example.f
```
2. Use the **ar** command to add the generated object files to an archive library file. For example:

```
ar -rv libfoo.a bar.o example.o
```

Compiling a shared library

To compile a shared library:

1. Compile your source files into an object file, with no linking. Note that in the case of compiling a shared library, the **-qpik** compiler option is also used. For example:

```
xlf -qpik -c foo.f
```
2. Use the **-qmkshrobj** compiler option to create a shared object from the generated object files. For example:

```
xlf -qmkshrobj -o libfoo.so foo.o
```

Related information in the *XL Fortran Compiler Reference*



-qpik



-qmkshrobj

Linking a library to an application

You can use the same command string to link a static or shared library to your main program. For example:

```
xlf -o myprogram main.f -Ldirectory [-Rdirectory] -lfoo
```

where *directory* is the path to the directory containing the library.

By using the **-l** option, you instruct the linker to search `libfoo.so` in the directory specified via the **-L** option (and, for a shared library, the **-R** option). If it is not found, the linker searches for `libfoo.a`. For additional linkage options, including options that modify the default behavior, see the operating system **ld** documentation.

Related information in the *XL Fortran Compiler Reference*



-l



-L



-R

Linking a shared library to another shared library

Just as you link modules into an application, you can create dependencies between shared libraries by linking them together. For example:

```
xlf -qmkshrobj -o mylib.so myfile.o -ldirectory -Rdirectory -lfoo
```

Related information in the XL Fortran Compiler Reference



-qmkshrobj



-R



-L

Compiling XL Fortran SMP programs

You can use the `xlf_r`, `xlf90_r`, `xlf95_r`, or `xlf2003_r` command to compile XL Fortran SMP programs. The `xlf_r` command is similar to the `xlf` command; the `xlf90_r` command is similar to the `xlf90` command; the `xlf95_r` command is similar to the `xlf95` command; the `xlf2003_r` command is similar to the `xlf2003` command. The main difference is that the components for multi-threading are used to link and bind the object files if you specify the `xlf_r`, `xlf90_r`, `xlf95_r`, or `xlf2003_r` command.

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

POSIX pthreads API support

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

To compile and then link your program with the standard interface libraries, use the `xlf_r`, `xlf90_r`, `xlf95_r`, or `xlf2003_r` command. For example, you could specify:

```
xlf95_r test.f
```

Compilation order for Fortran programs

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

Canceling a compilation

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

XL Fortran input files

The input files to the compiler are:

Source Files (.f or .F suffix)

All `.f`, `.f77`, `.f90`, `.f95`, `.f03`, and `.F`, `.F77`, `.F90`, `.F95`, and `.F03` 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`.

Related information: See “Passing Fortran files through the C preprocessor” on page 31.

The `fsuffix` and `cppsuffix` attributes in “Editing the default configuration file” on page 14 and “`-qsuffix`” on page 253 let you select a different suffix.

Object Files (.o suffix)

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

Related information: See “Linking” on page 90 and “Linking XL Fortran programs” on page 33.

The `osuffix` attribute, which is described in “Editing the default configuration file” on page 14 and “`-qsuffix`” on page 253, 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 “Editing the default configuration file” on page 14 and “`-qsuffix`” on page 253, lets you select a different suffix.

Shared Object or Library Files (.so suffix)

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

Configuration Files (.cfg suffix)

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

The default configuration file is `/opt/ibmcmp/xlf/13.1/etc/xlf.cfg`.

Related information: See “Using custom compiler configuration files” on page 10 and “`-F`” on page 102 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`” on page 104 and “`-qmoddir`” on page 201.

Profile Data Files

The `-qpdf1` option produces runtime 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 `“-qpdf1, -qpdf2”` on page 208.

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”` on page 110 for information on selecting a different name and `“-c”` on page 99 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”` on page 99 and “Linking XL Fortran programs” on page 33. For information on renaming the object file, see `“-o”` on page 110.

Assembler Source Files: `filename.s`

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

Related information: See `“-S”` on page 279 and “Linking XL Fortran programs” on page 33. For information on renaming the assembler source file, see `“-o”` on page 110.

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 “Listings, messages, and compiler information” on page 85.

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”` on page 201.

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” on page 100.

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 “-qpdf1, -qpdf2” on page 208.

Scope and precedence of option settings

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

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

If you specify an option more than once with different settings, the last setting generally takes effect. Any exceptions are noted in the individual descriptions in the Chapter 7, “Detailed descriptions of the XL Fortran compiler options,” on page 95 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
```

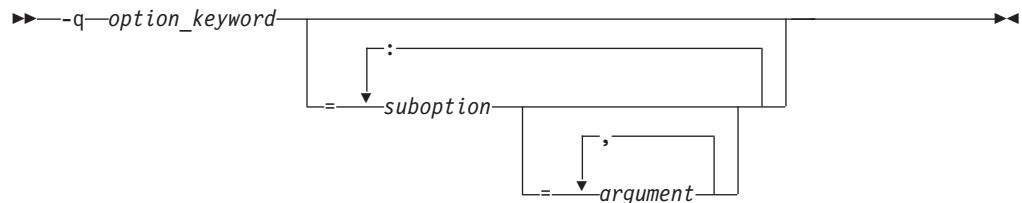
(There are some exceptions, such as **-pg**, which is a single option and not the same as **-p -g**.)

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:



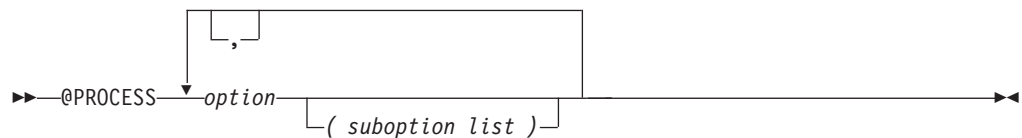
This format is more restrictive about the placement of blanks; you must separate individual **-q** options by blanks, and there must be no blank between a **-q** option and a following argument string. Unlike the names of flag options, **-q** option names are not case-sensitive except that the **q** 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 -O3 -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" command

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 `/opt/ibmcmp/xlf/13.1/etc/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 `/opt/ibmcmp/xlf/13.1/etc/xlf.cfg`:

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

and issue this command:

```
xlf95 -Wa,-Z -Wl,-s -w produces_warnings.s uses_many_symbols.f
```

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

Related information: See “-W” on page 284 and “Using custom compiler configuration files” on page 10.

Displaying information inside binary files (strings)

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

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

For example to see the information embedded in `/opt/ibmcmp/xlf/13.1/exe/xlfentry` issue the following command:

```
strings /opt/ibmcmp/xlf/13.1/exe/xlfentry | grep "@(#)"
```

Compiling for specific architectures

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

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

If the compiling machine is also the target architecture, **-qarch=auto** will automatically detect the setting for the compiling machine. For more information on this compiler option setting, see “-qarch” on page 120 and also **-O4** and **-O5** under the **-O** option.

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

Passing Fortran files through the C preprocessor

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

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

To call **cpp** for a particular file, use a file suffix of **.F**, **.F77**, **.F90**, **.F95**, or **.F03**. If you specify the **-d** option, each **.F*** file *filename.F** is preprocessed into an intermediate file **Ffilename.f**, which is then compiled. If you do not specify the **-d** option, the intermediate file name is `/tmpdir/F8xxxxxx`, where *x* is an alphanumeric character and *tmpdir* is the contents of the **TMPDIR** environment variable or, if you have not specified a value for **TMPDIR**, **/tmp**. You can save the intermediate file by specifying the **-d** compiler option; otherwise, the file is deleted. If you only want to preprocess and do not want to produce object or executable files, specify the **-qnoobject** option also.

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

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

```

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

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

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

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

end

```

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

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

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

Related information:

- “-d” on page 100
- “-t” on page 280
- “-W” on page 284
- “-qfpp” on page 157
- “-qppsuborigarg” on page 219
- “Using custom compiler configuration files” on page 10

cpp directives for XL Fortran programs

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

Passing options to the C preprocessor

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

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

Avoiding preprocessing problems

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

Consider the following example:

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

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

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

Linking XL Fortran programs

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

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

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

After linking, follow the instructions in “Running XL Fortran programs” on page 35 to execute the program.

To link a library, follow the instructions in “Compiling and linking a library” on page 24.

Compiling and linking in separate Steps

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

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

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

Passing options to the ld command

For the detailed information about passing options to the `ld` command, see “Passing command-line options to the `ld` or `as` command” on page 30.

Dynamic and static linking

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

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

Dynamically linked programs take up less disk space and less virtual memory if more than one program uses the routines in the shared libraries. During linking, there are less chances for naming conflicts with library routines or external data objects because only exported symbols are visible outside a shared library. They may perform better than statically linked programs if several programs use the same shared routines at the same time. They also allow you to upgrade the routines in the shared libraries without relinking.

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

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

Statically linked programs can be moved to and run on systems without the XL Fortran libraries. They may perform better than dynamically linked programs if they make many calls to library routines or call many small routines. There are more chances for naming conflicts with library routines or external data objects because all global symbols are visible outside a static library. They also may not work if you compile them on one level of the operating system and run them on a different level of the operating system.

To link statically, add the `-qstaticlink` option to the linker command. For example:

```
xlf95 -qstaticlink test.f
```

Avoiding naming conflicts during linking

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

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

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

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

- Link your programs dynamically, which is the default.

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

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

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 file along with any runtime arguments on the command line.

Canceling execution

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

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

Compiling and executing on different systems

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

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

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

Related information: See “Dynamic and static linking” on page 34.

Runtime libraries for POSIX pthreads support

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

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

Cmd.	Libraries Used	Include Directory
xlf90_r xlf95_r xlf_r	/opt/ibmcmp/lib/libxlf90_r.so /opt/ibmcmp/lib64 /libxlf90_r.so /opt/ibmcmp/lib/libxlsmp.so /opt/ibmcmp/lib64/libxlsmp.so	/opt/ibmcmp/xlf/13.1/include

Selecting the language for runtime messages

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

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

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

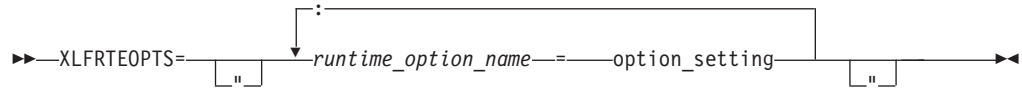
Related information: See “Environment variables for national language support” on page 8.

Setting runtime options

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

The XLFRTOPTS environment variable

The **XLFRTOPTS** environment variable allows you to specify options that affect the runtime behavior of items such as I/O, EOF error-handling, the specification of random-number generators, and more. You can declare **XLFRTOPTS** 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 **XLFRTOPTS** 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 runtime error message.
- A **DEALLOCATE** statement needs to issue a runtime error message.
- The multi-threaded implementation of the **MATMUL** procedure is executed.

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

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

You can specify the following runtime options with the **XLFRTOPTS** environment variable or the **SETRTEOPTS** procedure:

aggressive_array_io={yes | no}

Controls whether or not the XL Fortran run time will take advantage of descriptor information when deciding to apply slower or faster algorithms to do array I/O operations. Descriptor information that specifies an array or array section as contiguous can be used to apply the faster algorithms which would otherwise be unsafe if the array or array section was not contiguous. The default is to perform aggressive array I/O operations.

Code executing under the current XL Fortran runtime but compiled with older XL Fortran compilers can cause the aggressive array I/O operations to be unsafe if the older compilers did not set the XL Fortran descriptor information correctly. This can be a problem with code built with old XL Fortran compilers no longer in service or built with XL Fortran compilers not at the latest service levels. Older code should be recompiled, if possible, with the current compiler instead of relying on the use of this option.

buffering={enable | disable_preconn | disable_all}

Determines whether the XL Fortran runtime 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** runtime option to disable the buffering in the XL Fortran runtime 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 flush the contents of the I/O buffer for that logical unit with the **FLUSH** statement or the Fortran service routine **flush_**.

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

enable

The Fortran runtime library maintains an I/O buffer for each connected logical unit. The current read-write file pointers that the runtime library maintains might not be synchronized with the read-write pointers of the corresponding files in the file system.

disable_preconn

The Fortran runtime 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 runtime library maintains for the preconnected units are the same as the read-write pointers of the corresponding files in the file system.

disable_all

The Fortran runtime library does not maintain I/O buffers for any logical units. You should not specify the **buffering=disable_all** option with Fortran programs that perform asynchronous I/O.

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

Fortran main program:

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

C subroutine (c_func.c):

```

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

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

```

Input data file (infile):

```

11111
22222
33333
44444

```

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

```
$ main < infile
```

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

```

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

```

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

buffer_size=*size*

Specifies the size of I/O buffers in bytes instead of using the block size of devices. *size* must be either -1 or an integer value that is greater than or equal to 4096. The default, -1, uses the block size of the device where the file resides.

Using this option can reduce the amount of memory used for I/O buffers when an application runs out of memory because the block size of devices is very large and the application opens many files at the same time.

Note the following when using this runtime option:

- Preconnected units remain unaffected by this option. Their buffer size is the same as the block size of the device where they reside except when the block size is larger than 64KB, in which case the buffer size is set to 64KB.
- This runtime option does not apply to files on a tape device or logical volume.
- Specifying the buffer size with the **SETRTEOPTS** procedure overrides any value previously set by the **XLFRTEOPTS** environment variable or **SETRTEOPTS** procedure. The resetting of this option does not affect units that have already been opened.

cnvrr={yes | no}

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

Related information: For more information about conversion errors, see *Data transfer statements* in the *XL Fortran Language Reference*. For more information about **IOSTAT** values, see *Conditions and IOSTAT values* in the *XL Fortran Language Reference*.

cpu_time_type={usertime | systime | alltime | total_usertime | total_systime | 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.

systime

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_systime

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

total_alltime

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

default_recl={64 | 32}

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

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

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

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

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

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

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

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

errloc={yes | no}

Controls whether the file name and line number appear in runtime error messages when a runtime error condition occurs during I/O or an **ALLOCATE/DEALLOCATE** statement and an error message is issued. By default, the line number and file name appear prepended to the runtime error

messages. If **errloc=no** is specified, runtime error messages are displayed without the source location information.

The **errloc** runtime option can be specified with the **SETRTEOPTS** procedure, as well.

erroeof={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 runtime 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.

errthrdnum={yes | no}

When **errthrdnum=yes** is in effect, the thread number of the executing OpenMP thread specified by the **omp_get_thread_num** routine is appended to any XL Fortran runtime error messages generated. For single-threaded programs, the thread number will be 0. If **errloc=yes** is specified, the thread number appears in front of the file name and line number information. If the **IOMSG=** specifier is present in an I/O statement, the thread number is prepended in the message assigned to the variable specified by this option in the same format as displayed on standard error.

iostat_end={extended | 2003std}

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

extended

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

2003std

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

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

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

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

intrinths={num_threads}

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

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

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

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

90std Specifies that the compiler should flag any extensions to the Fortran 90 standard I/O statements and formats as errors.

95std Specifies that the compiler should flag any extensions to the Fortran 95 standard I/O statements and formats as errors.

2003std

Specifies that the compiler should accept all standard I/O statements and formats that the Fortran 95 standard specifies, as well as those Fortran 2003 formats that XL Fortran supports. Anything else is flagged as an error.

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

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

extended

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

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

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

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

```
program test1

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

10 format(I3)

write(1,fmt=10) 123

end
```

Specifying **langlvl=95std** results in a runtime 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 runtime error message is issued. However, if you specify **langlvl=90std**, a runtime error message is issued.

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

multconn={yes | no}

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

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

- Files have been connected for write-only (**ACTION='WRITE'**)
- Asynchronous I/O
- Files on sequential-access devices (such as pipes, terminals, sockets)

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

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

multconnio={tty | nulldev | combined | no }

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

combined

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

nulldev

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

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

Note: Using this option can produce unpredictable results.

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

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

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

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

namelist={new | old}

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

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

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

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

naninfoutput={2003std | old | default}

Controls whether the display of IEEE exceptional values conform to the Fortran 2003 standard or revert to the old XL Fortran behavior. This runtime option allows object files created with different compilation commands to output all IEEE exceptional values based on the old behavior, or the Fortran 2003 standard. The suboptions are:

default

Exceptional values output depends on how the program is compiled.

old

Exceptional values output conforms to the old XL Fortran behavior.

2003std

Exceptional values output conforms to the Fortran 2003 standard.

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** runtime 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** runtime option to **yes**, and set the environment variable **XLFSCRATCH_unit** to the name of the file you want to be associated with the specified unit number. See *Naming scratch files* in the *XL Fortran Optimization and Programming Guide* for examples.

ufmt_littleendian={units_list}

Specifies unit numbers of unformatted data files on which little-endian I/O is to be performed. The little-endian format data in the specified unformatted files is converted, on-the-fly, during the I/O operation to and from the big-endian format used on machines where XL Fortran applications are running.

This runtime option does not work with internal files; internal files are always **FORMATTED**. Units specified must be connected by an explicit or implicit **OPEN** for the **UNFORMATTED** form of I/O.

The syntax for this option is as follows:

```
ufmt_littleendian=units_list
```

where:

```
units_list = units | units_list, units
```

```
units = unit | unit- | -unit | unit1-unit2
```

The unit number must be an integer, whose value is in the range 1 through 2 147 483 647.

unit Specifies the number of the logical unit.

unit- Specifies the range of units, starting from unit number *unit* to the highest possible unit number

-unit Specifies the range of units, starting from unit number 1 to unit number *unit*.

unit1-unit2

Specifies the range of units, starting from unit number *unit1* to unit number *unit2*.

Note:

1. The byte order of data of type **CHARACTER** is not swapped.
2. The compiler assumes that the internal representation of values of type **REAL*4** or **REAL*8** is IEEE floating-point format compliant. I/O may not work properly with an internal representation that is different.

3. The internal representation of values of type **REAL*16** is inconsistent among different vendors. The compiler treats the internal representation of values of type **REAL*16** to be the same as XL Fortran's. I/O may not work properly with an internal representation that is different.
4. Conversion of derived type data is not supported. The alignment of derived types is inconsistent among different vendors.
5. Discrepancies in implementations from different vendors may cause problems in exchanging the little-endian unformatted data files between XL Fortran applications running on Linux and Fortran applications running on little-endian systems. XL Fortran provides a number of options that help users port their programs to XL Fortran. If there are problems exchanging little-endian data files, check these options to see if they can help with the problem.

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 runtime 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* in the *XL Fortran Optimization and Programming Guide* for examples.

uwidth={32 | 64}

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

xrf_messages={yes | no}

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

The following examples set the **cnvrr** runtime option to **yes** and the **xrf_messages** option to **no**.

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

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

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

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

Setting OMP and SMP run time options

The **XLSMPOPTS** environment variable allows you to specify options that affect SMP execution. The OpenMP environment variables, **OMP_DYNAMIC**, **OMP_NESTED**, **OMP_NUM_THREADS**, and **OMP_SCHEDULE**, allow you to control the execution of parallel code. For details on using these, see *XLSMPOPTS* and *OpenMP environment variables* sections in the *XL Fortran Optimization and Programming Guide*.

BLAS/ESSL environment variable

By default, the `libxlopt` library is linked with any application you compile with XL Fortran. However, if you are using a third-party Basic Linear Algebra Subprograms (BLAS) library or want to ship a binary that includes ESSL routines, you must specify these using the `XL_BLAS_LIB` environment variable. For example, if your own BLAS library is called `libblas`, set the environment variable as follows:

```
export XL_BLAS_LIB=/usr/lib/libblas.a
```

When the compiler generates calls to BLAS routines, the ones defined in the `libblas` library will be used at runtime instead of those defined in `libxlopt`.

XL_F_USR_CONFIG

Use the `XL_F_USR_CONFIG` environment variable to specify the location of a custom configuration file to be used by the compiler. The file name must be given with its absolute path. The compiler will first process the definitions in this file before processing those in the default system configuration file, or those in a customized file specified by the `-F` option; for more information, see “Using custom compiler configuration files” on page 10.

Other environment variables that affect runtime behavior

The `LD_LIBRARY_PATH`, `LD_RUN_PATH`, and `TMPDIR` environment variables have an effect at run time, as explained in “Correct settings for environment variables” on page 8. They are not XL Fortran runtime options and cannot be set in either `XLFRTEOPTS` or `XLSPMPOPTS`.

XL Fortran runtime exceptions

The following operations cause runtime exceptions in the form of `SIGTRAP` signals, which typically result in a “Trace/breakpoint trap” message:

- Character substring expression or array subscript out of bounds after you specified the `-C` option at compile time.
- Lengths of character pointer and target do not match after you specified the `-C` option at compile time.
- The flow of control in the program reaches a location for which a semantic error with severity of `S` was issued when the program was compiled.
- Floating-point operations that generate NaN values and loads of the NaN values after you specify the `-qfloat=nanq` option at compile time.
- Fixed-point division by zero.
- Calls to the TRAP hardware-specific intrinsic procedure.

The following operations cause runtime exceptions in the form of `SIGFPE` signals:

- Floating-point exceptions provided you specify the appropriate `-qfltrap` 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:

- “-C” on page 98
- “-qfltrap” on page 158
- “-qsigtrap” on page 234

Also see the following topics in the *XL Fortran Optimization and Programming Guide*:

- *Detecting and trapping floating-point exceptions* for more details about these exceptions
- *Controlling the floating-point status and control register* for a list of exception handlers.

Chapter 5. Tracking and reporting compiler usage

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

To use this feature, follow these steps:

1. Understand how the feature works. See “Understanding utilization tracking and reporting” for more information.
2. Investigate how the compiler is used in your organization, and decide how you track the compiler usage accordingly. See “Preparing to use this feature” on page 58 for more information.
3. Configure and enable utilization tracking. See “Configuring utilization tracking” on page 63 for more information.
4. Use the utilization reporting tool to generate usage reports or prune usage files. See “Generating usage reports” on page 71 or “Pruning usage files” on page 74 for more information.

Understanding utilization tracking and reporting

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

Overview

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

You can use the utilization tracking and reporting feature in various ways based on how the compiler is used in your organization. “Four usage scenarios” on page 50 illustrates the typical usage scenarios of this feature.

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

Utilization tracking

A utilization tracking configuration file `urtxlf1301linux.cfg` is included in the default compiler installation. You can use this file to enable utilization tracking and control different aspects of the tracking.

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

For more information, see “Configuring utilization tracking” on page 63.

Note: Utilization tracking is disabled by default.

Utilization reporting tool

The utilization reporting tool generates compiler usage reports based on the information in the usage files. You can optionally prune the usage files with the tool. For more information, see “Generating usage reports” on page 71 and “Pruning usage files” on page 74.

Four usage scenarios

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

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

Note: Actual usage is not limited to these scenarios.

“Scenario: One machine, one shared .cuf file”

“Scenario: One machine, multiple .cuf files” on page 52

“Scenario: Multiple machines, one shared .cuf file” on page 54

“Scenario: Multiple machines, multiple .cuf files” on page 56

Scenario: One machine, one shared .cuf file

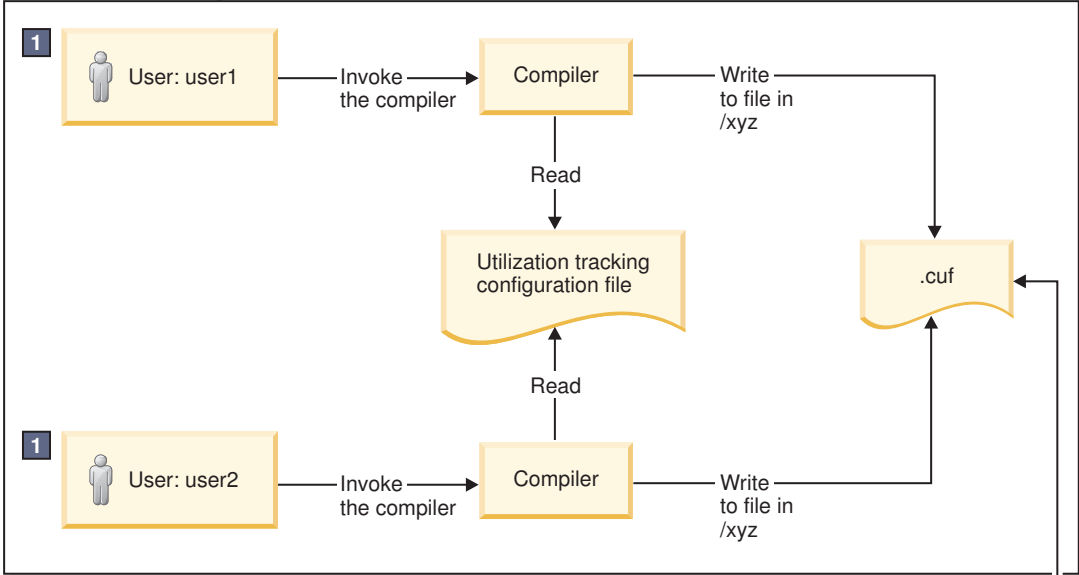
This scenario describes an environment where all the compilations are done on one machine and all users share one .cuf file.

The advantage of using the approach in this scenario is that it simplifies report generation and usage file pruning, because the utilization report tool only need to access one .cuf file. The disadvantage is that all compiler users need to compete for access to this file. Because the file might become large, it might have an impact on performance. Some setup work is also required to create the shared .cuf file and to give all compiler users write access. The “The number of usage files” on page 61 section provides detailed information about using a single usage file for all compiler users.

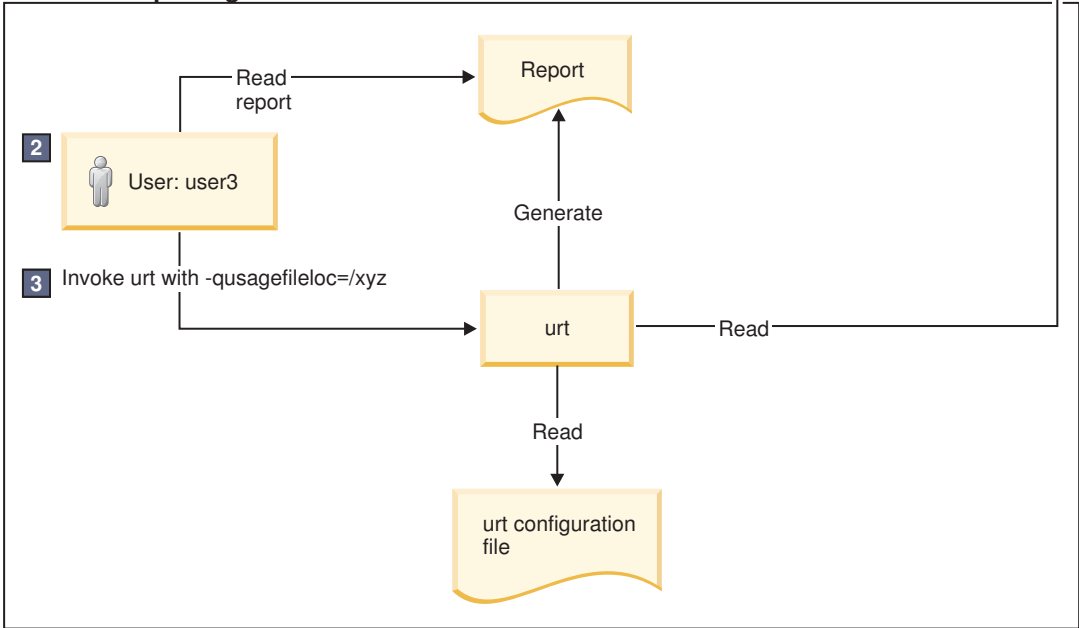
In this scenario, two compiler users run the compiler on the same machine and their utilization information is recorded in a shared .cuf file. The utilization tracking configuration file for the compiler is modified to point to the location of the .cuf file. When the compiler is invoked, it writes the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

The following diagram illustrates this scenario.

Utilization tracking



Utilization reporting



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

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

The diagram reflects the following points:

1. user1 and user2 use the same utilization tracking configuration file, which manages the tracking functionality centrally. A common location /xyz is created to keep a shared .cuf file.
2. When user1 and user2 invoke the compiler, the utilization information is recorded in the .cuf file under the common directory /xyz.

3. user3 invokes `urt` with `-qusagefileloc=/xyz` to generate usage reports.

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

Scenario: One machine, multiple .cuf files

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

The approach in this scenario has the following advantages:

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

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

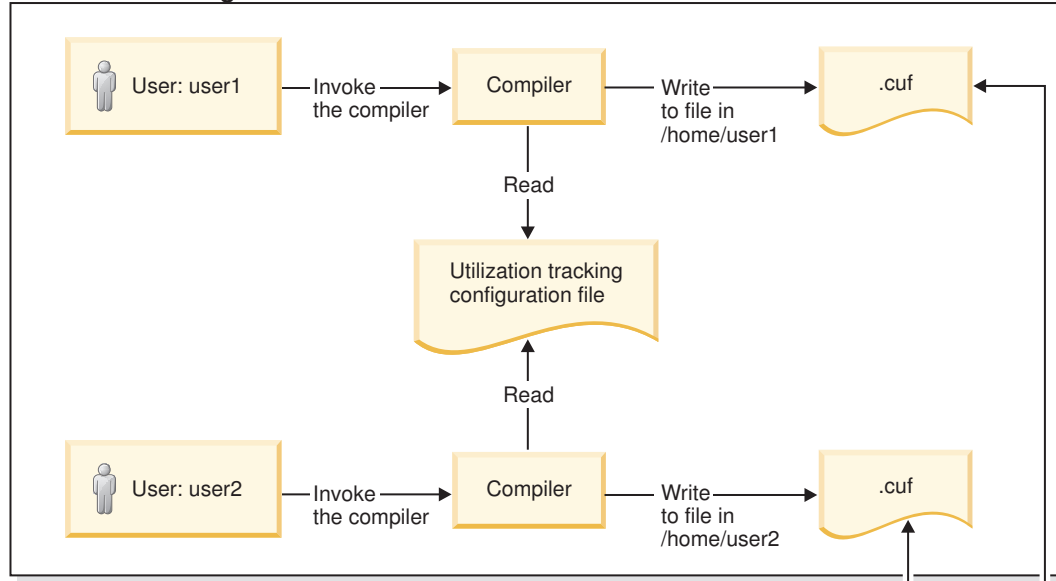
- Compiler users might not know that the file has been created or what it is when they see the file. In this case, they might delete the file.
- Some users' home directories might be on file systems that are mounted from a remote system. This causes utilization tracking to use a remote file, which might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

Instead of using each user's home directory, the `.cuf` files for each user can be created in a common location. The “Usage file location” on page 60 section provides detailed information about how to create these files in a common location.

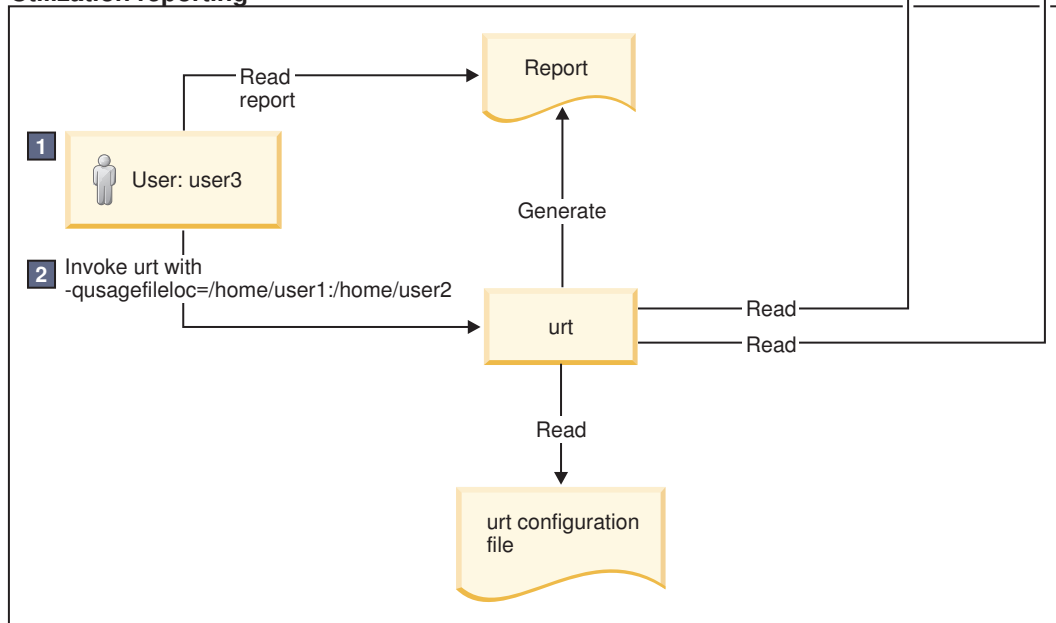
In this scenario, two compiler users run the compiler on the same machine and they have their own `.cuf` files. When the compiler is invoked, it automatically creates a `.cuf` file for each user and writes the utilization information to that file. You can then use the utilization reporting tool to retrieve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

The following diagram illustrates this scenario.

Utilization tracking



Utilization reporting



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

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

This diagram reflects the following points:

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

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

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

In this case, **urt** looks for `.cuf` files in all users' home directories.

Scenario: Multiple machines, one shared .cuf file

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

The advantage of the approach in this scenario is that using one `.cuf` file can simplify the report generation and the usage file pruning process. The section “The number of usage files” on page 61 provides detailed information about using a single usage file for all compiler users. The `.cuf` file is already on the machine where the utilization reporting tool is installed. You do not need to copy the file to that machine or install the tool on multiple machines to prune the `.cuf` files.

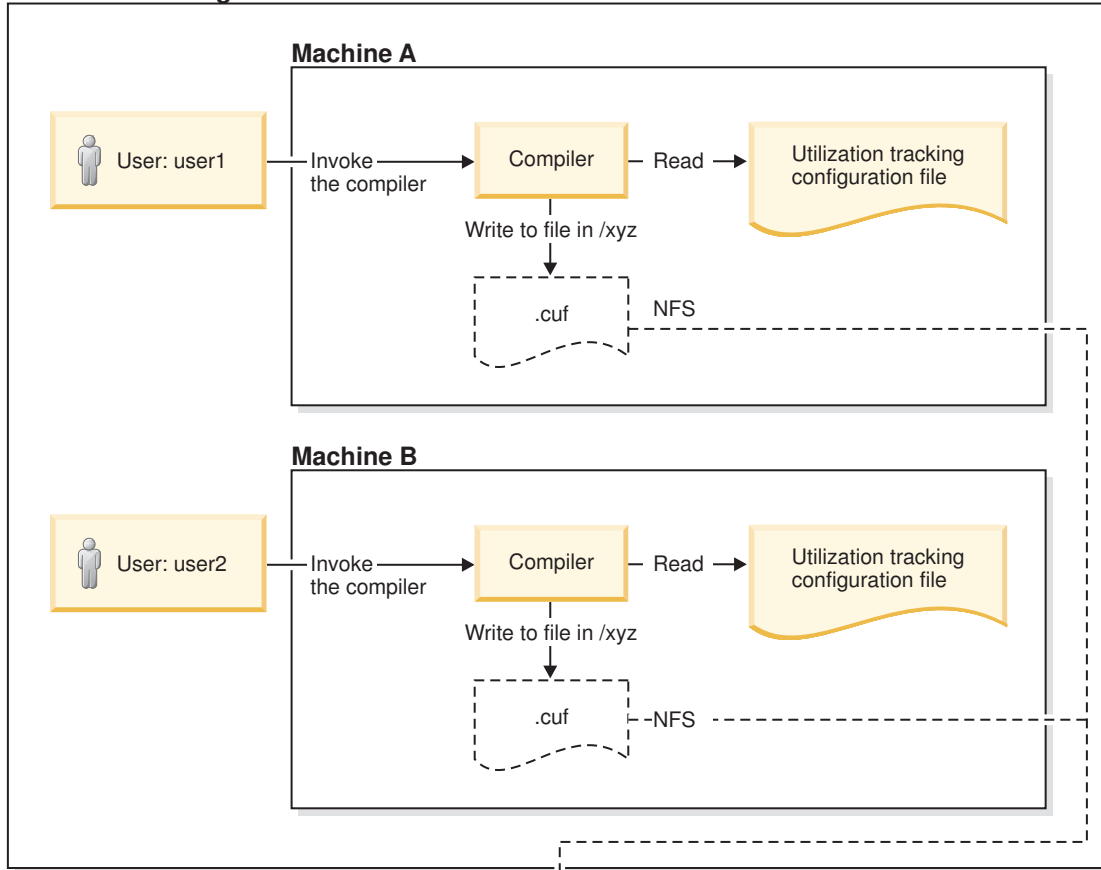
This approach has the following disadvantages:

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

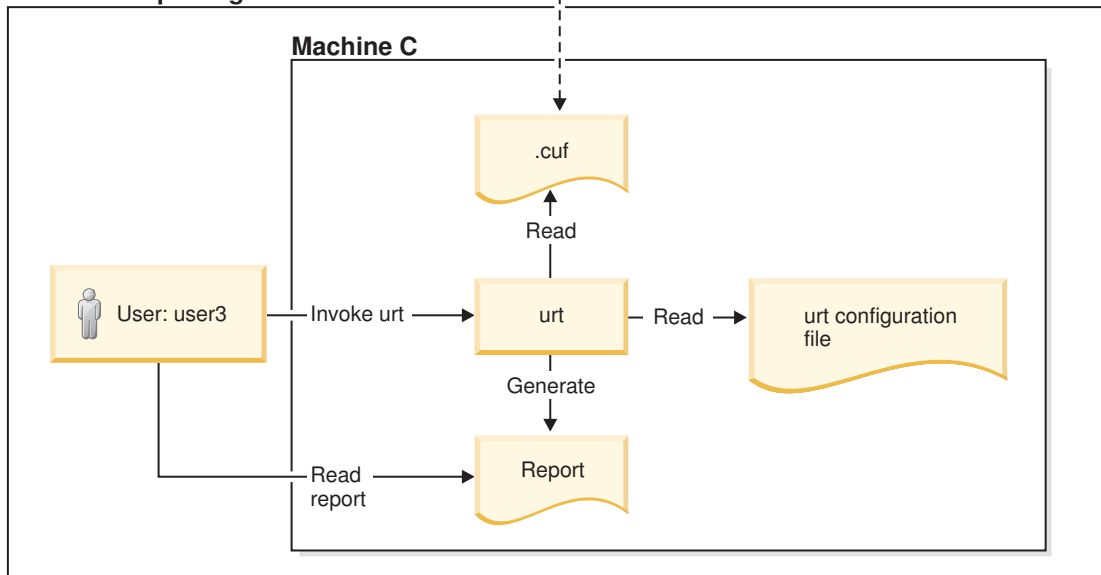
In this scenario, two compiler users run the compilers on separate machines and they use one shared `.cuf` file on a network file system, such as NFS, DFS, or AFS. When the compiler is invoked, it writes the utilization information to that file. You can then use the utilization reporting tool to retrieve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

The following diagram illustrates this scenario.

Utilization tracking



Utilization reporting



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

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

This diagram reflects the following points:

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

Notes:

- Although each machine has its own configuration file, the contents of these files must be the same.
 - Centrally managing the utilization tracking functionality can reduce your configuration effort and eliminate possible errors. The “Central configuration” on page 59 section provides detailed information about how you can use a common configuration file shared by compiler users using different machines.
2. A network file system is set up for the central management of the .cuf files. When user1 and user2 invoke the compilers from Machine A and Machine B, the utilization information of both compilers is written to the .cuf file on Machine C.
 3. user3 invokes **urt** to generate usage reports from the .cuf file on Machine C.

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

Scenario: Multiple machines, multiple .cuf files

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

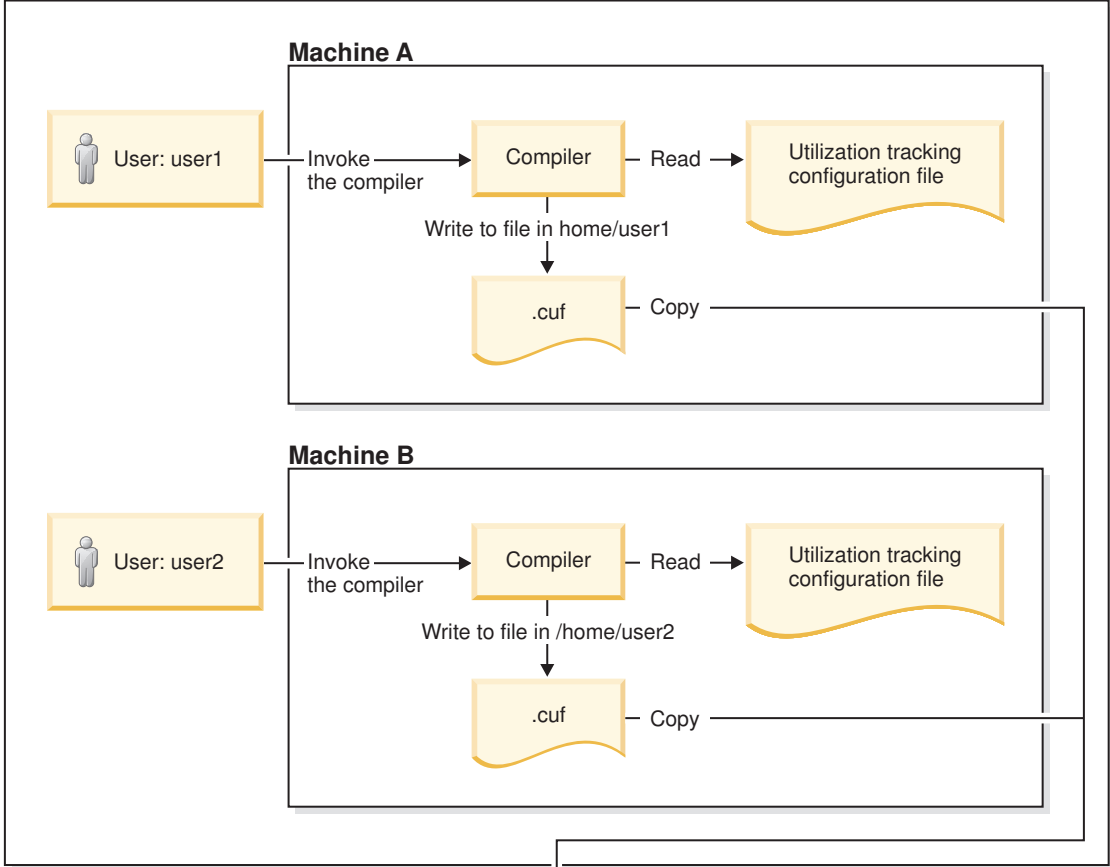
In this scenario, two compiler users run the compilers on separate machines and they have their own .cuf files. When the compiler is invoked, it writes the utilization information to that file. You can then use the utilization reporting tool to retrieve the utilization information from the file and generate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

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

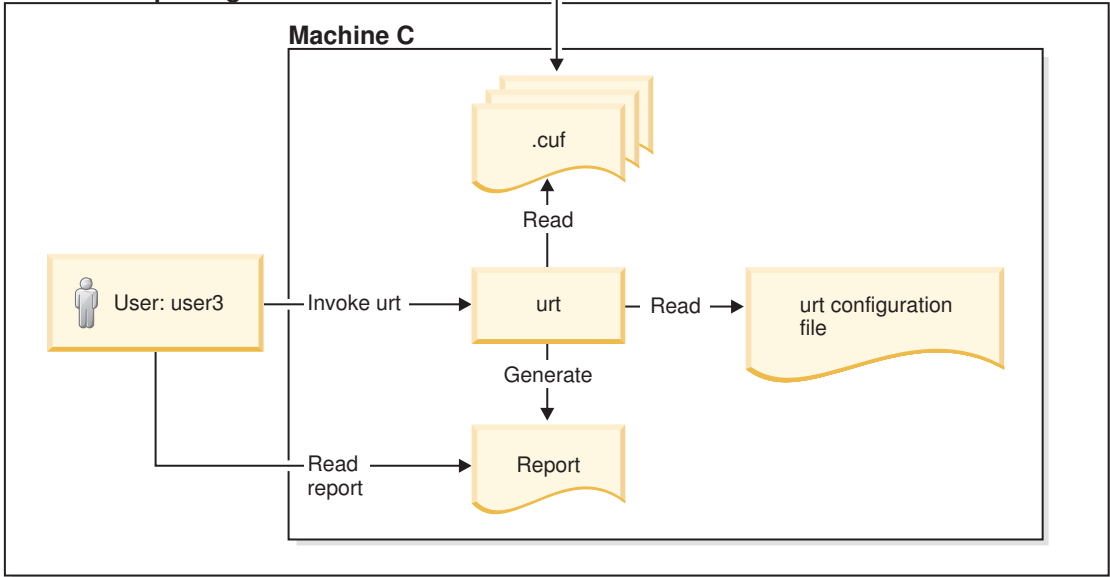
- Use a network file system, such as NFS, DFS, or AFS.
- Copy the files from their original locations to the machine where you plan to run the utilization reporting tool. You can use **ftp**, **rtp**, **rsync** or any other remote copy command to copy the files.

The following diagram illustrates this scenario.

Utilization tracking



Utilization reporting



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

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

This diagram reflects the following points:

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

Notes:

- Although each machine has its own configuration file, the contents of these files must be the same.
 - Centrally managing the utilization tracking functionality can reduce your configuration effort and eliminate possible errors. The “Central configuration” on page 59 section provides detailed information about how you can use a common configuration file shared by compiler users using different machines.
2. When user1 and user2 invoke the compilers, the utilization information is recorded in the two .cuf files under their respective home directories, /home/user1 and /home/user2.

Note: These .cuf files can also be created in another common location, for example, /var/tmp. The “Usage file location” on page 60 section provides detailed information about how to create these files in a common location.

3. user3 copies the two .cuf files from Machine A and Machine B to Machine C.
4. user3 invokes **urt** to generate usage reports from the .cuf files on Machine C.

Related information

- “Preparing to use this feature”
- “Configuring utilization tracking” on page 63
- “Generating usage reports” on page 71
- “Pruning usage files” on page 74

Preparing to use this feature

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

The following sections describe those considerations in detail:

Time synchronization

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

The usage report generated by the utilization reporting tool lists the time when the compiler invocations start and end. The report also determines which invocations are concurrent. This information is much less reliable and useful if time is not synchronized across these machines.

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

License types and user information

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

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

- The number of Concurrent User licenses that you have for this compiler. This information is required for the **-qmaxconcurrentusers** entry in the utilization tracking configuration file.

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

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

Central configuration

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

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

If you have multiple installations of the compiler, you need to maintain a single utilization tracking config file and reference it from all installations. Any changes you make to the utilization tracking configuration file, including enabling or disabling utilization tracking, can automatically apply to all compiler installations when users invoke the compiler. In each installation, there is a symlink named `urt_client.cfg`, located under `/opt/ibmcmp/xlf/13.1/urt`. Modify the symlink to point to this shared instance of the configuration file.

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

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

Concurrent user considerations

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

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

- Users who have their own Authorized User license are considered exempt users, because their use of the compiler does not consume any Concurrent User licences.

- Users who have multiple accounts. Because the accounts belong to the same user, invocations of the compiler while logged on using those accounts are counted as usage by a single user.

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

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

Notes:

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

Usage file considerations

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

Usage file location

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

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

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

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

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

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

This creates a .cuf file for each user in the specified location, such as user1.cuf or user2.cuf. It is easier to run the utilization reporting tool to generate the usage

report from the .cuf files in this central location. You only need to pass the path of the location, /var/tmp/track_compiler_use to the utilization reporting tool , and then the tool can read all the .cuf files in that location.

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

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

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

The number of usage files

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

Using separate usage files for different compiler users

The advantages of using separate usage files are as follows:

- It might provide better performance because compiler users access their own usage files instead of competing for access to a shared one and separate usage files are usually smaller.
- Usage file for a user can be automatically created when the user uses the compiler to compile a program. There is no need to explicitly create a usage file for each user beforehand. For more information, see “Usage file location” on page 60.
- When generating utilization reports, you usually include all compiler users. However, if there are circumstances in which you want to exclude some users, you can simply omit their usage files when you invoke the utilization reporting tool. For example, you might want to omit users who have their own Authorized User license.

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

Using a single usage file for all compiler users

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

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

Usage files on multiple machines

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

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

- Copy the usage files from the machines where the compiler is used to the machine where the utilization reporting tool is installed. You can use any remote copy command, for example, **ftp**, **rnp**, **scp**, and **rsync**. In this case, the usage files are being accessed locally by both the compiler, for utilization tracking, and by the utilization reporting tool, for generating the usage report. Accessing the files locally yields the best performance.
- Use a distributed file system to export the file system from the machines where the compiler is used, and mount those file systems on the machine where the utilization reporting tool is installed. When you run the utilization reporting tool, it can access the usage files remotely via the mounted file systems.
- You can also export the file system from the machine where the utilization reporting tool is installed, and mount that file system on each machine where the compiler is used, using it as the location of the usage files where the compiler is recording its utilization. In this approach, the compiler records utilization in a remote usage file, and the utilization reporting tool reads the usage file locally.

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

Usage file size

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

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

For more information about how to prune files, see “Pruning usage files” on page 74.

Regular utilization checking

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

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

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

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

Testing utilization tracking

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

Testing with a limited number of users

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

The default compiler configuration file, `xlfcfg.$OSRelease.gcc$gccVersion`. For example, `xlfcfg.sles11.gcc432` or `xlfcfg.rhel5.5.gcc412` contains two entries, `xlurt_cfg_path` and `xlurt_cfg_name`, which specify the location of the utilization tracking configuration file. You need to perform the following tasks to let the specified users use the separate utilization tracking configuration file:

1. Create a separate compiler configuration file or stanza, in which the `xlurt_cfg_path` and `xlurt_cfg_name` entries specify the location of the utilization tracking configuration file you want to use.
2. Ask these users to use the following compiler option or environment variable to instruct the compiler to use the separate compiler configuration file or stanza, which in turn allows them to use the separate utilization tracking configuration file.
 - The `-F` option
 - The “`XLF_USR_CONFIG`” on page 47 environment variable

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

Testing with a separate compiler installation

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

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

Related information

- “Configuring utilization tracking”

Configuring utilization tracking

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

The default location of the configuration file is `/opt/ibmcmp/xlf/13.1/urt` and its file name is `urtxlf1301linux.cfg`.

The compiler uses a symlink to specify the location of the utilization tracking configuration file. The symlink is also located in `/opt/ibmcmp/xlf/13.1/urt` and its name is `urt_client.cfg`. In the following situations, you might need to change the symlink:

- If you want to use a utilization tracking configuration file in a different location, change the symlink to point to this location.
- If you have multiple installations of the same compiler, and you plan to use a single utilization tracking configuration file, change the symlink in each installation to point to that file. For more information, see “Central configuration” on page 59.

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

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

Editing utilization tracking configuration file entries

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

The entries are divided into two categories.

1. The entries in the *Product information* category identify the compiler. Do not modify these entries.
2. The entries in the *Tracking configuration* category can be used to configure utilization tracking for this product. Changes to these entries take effect in the usage file upon the next compiler invocation. In this case, the compiler emits a message to indicate that the new configuration values have been saved in the usage file. When you generate a report from the usage file, the new values are used.

The following rules apply when you modify the entries:

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

Notes:

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

Product information

-qprodId=*product_identifier_string*

Indicates the unique product identifier string.

-qprodVer=*product_version*

Indicates the product version.

-qprodRel=*product_release*

Indicates the product release.

-qprodName=*product_name*

Indicates the product name.

-qconcurrentusagescope=*prod | ver | rel*

Specifies the level at which concurrent users are counted, and their numbers are limited. The suboptions are as follows:

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

Default: **-qconcurrentusagescope=prod**

Tracking configuration

-qmaxconcurrentusers=*number*

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

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

Default: 0

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

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

user_account_info can be any combination of the following items:

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

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

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

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

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

This is equivalent to the previous examples.

Note: Specifying this entry does not exempt users from compiler utilization tracking. It only exempts them from being counted as concurrent users. To optimize utilization tracking performance, the format of the specified value is not validated until the report is produced. For more information about counting concurrent users, see “Concurrent user considerations” on page 59.

-qqualhostname | -qnoqualhostname

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

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

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

-qenabletracking | -qnoenabletracking

Enables or disables utilization tracking.

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

-qusagefileloc=directory_or_file_name

Specifies the location of the usage file.

By default, a `.cuf` file is automatically created for each user in their home directory. You can set up a central location where the files for each user can be created. For more information, see “Usage file location” on page 60.

The following rules apply when you specify this entry:

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

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

You can use the following variables for this option:

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

-qfileaccessmaxwait=number_of_milliseconds

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

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

Default: 3000 milliseconds

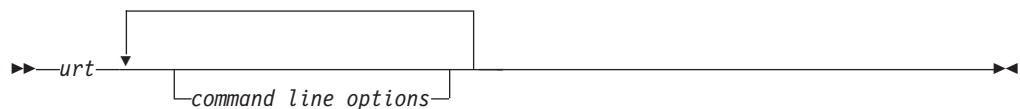
Notes:

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

Understanding the utilization reporting tool

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

The tool is located in the `/opt/ibmurt/1.1/bin` directory. You can use the **urt** command to invoke it. The syntax of the **urt** command is as follows:



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

Command-line options control how usage reports are generated. For more information about the options, see “Utilization reporting tool command-line options” on page 68.

A default configuration file `ibmurt.cfg` is provided in the `/opt/ibmurt/1.1/config` directory. Entries in this file take the same form as the command-line options and have the same effect. You can also create additional configuration files and use the **-qconfigfile** option to specify their names.

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

1. The default configuration file
2. The additional configuration file specified with **-qconfigfile**
3. The command line

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

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

Utilization reporting tool command-line options

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

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

-qreporttype=detail | maxconcurrent

Specifies the type of the usage report to generate.

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

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

Default: **-qreporttype=maxconcurrent**.

-qrptmaxrecords=num | nomax

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

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

-qusagefileloc=directory_or_file_name

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

The following rules apply when you specify this option:

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

Note:

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

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

-qmaxsubdirs=num | nomax

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

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

Default: 0.

-qconfigfile=*file_path*

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

For more information about how the utilization reporting tool uses the configuration file, see “Understanding the utilization reporting tool” on page 67.

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

-qsameuser=*user_account_info*

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

user_account_info can be any combination of the following items:

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

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

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

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

or

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

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

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

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

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

The following rules apply when you specify this option:

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

The following examples illustrates the differences:

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

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

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

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

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

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

Note: Specifying this option does not prevent user information from being listed in the usage report. For more information about concurrent users, see “Concurrent user considerations” on page 59.

-qadjusttime=*time_adjustments*

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

The following rules apply when you use this option:

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

For example:

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

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

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

Notes:

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

-qusagefilemaxage=*number_of_days* | nomax

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

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

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

-qusagefilemaxsize=*number_of_MB* | nomax

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

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

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

-qtimesort=ascend | descend

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

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

Default: **-qtimesort=ascend**.

Generating usage reports

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

To generate a report, use the command-line options or the **urt** configuration file to specify how you want a report to be generated. For more information about these options, see “Utilization reporting tool command-line options” on page 68.

Notes:

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

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

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

- Exit code = "1".

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

- Exit code ="0".

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

For more information about the **urt** command, see "Understanding the utilization reporting tool" on page 67.

Understanding usage reports

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

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

1. The date and time when the report is generated.
2. The .cuf file or a list of all .cuf files used to generate the report.
3. The options that have been passed to the **urt** command, with default values for any unspecified options.
4. Possible messages categorized as ERROR, WARNING, or INFO. For detailed information about possible messages, see "Diagnostic messages from utilization tracking and reporting" on page 75.

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

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

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

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

```
DATE: 12/18/09      TIME: 01:30:24
```

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

```
reporttype=detail
maxsubdirs=0
configfile="/opt/ibmurt/1.1/config/ibmurt.cfg"
rptmaxrecords=nomax
*adjusttime=
usagefileloc="/home/testrun/ibmxlcompiler.cuf"
*sameuser=
timesort=ascend
usagefilemaxsize=nomax
usagefilemaxage=nomax
```

```
FILES USED:
```

```
/home/testrun/ibmxlcompiler.cuf
```

```
REPORT DETAILS
-----
```


USAGE INFORMATION FOR PRODUCT: IBM XL Fortran for Linux 13.1

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

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

Exempt Users:

Product invocations:

Start Time	End Time	User	Number of Concurrent Users
12/17/09 16:56:44	12/17/09 16:57:13	user1@host1.ibm.com	1
12/18/09 00:58:29	12/18/09 00:58:32	user2@host2.ibm.com	1
12/18/09 01:16:01	12/18/09 01:16:02	user3@host3.ibm.com	5 (exceeds max. allowed)
12/18/09 01:16:02	12/18/09 01:16:26	user2@host2.ibm.com	5 (exceeds max. allowed)
12/18/09 01:16:08	12/18/09 01:16:08	user3@host2.ibm.com	5 (exceeds max. allowed)
12/18/09 01:16:12	12/18/09 01:16:12	user2@host1.ibm.com	5 (exceeds max. allowed)
12/18/09 01:16:24	12/18/09 01:16:28	user1@host2.ibm.com	5 (exceeds max. allowed)
12/18/09 01:26:11	12/18/09 01:27:46	user3@host3.ibm.com	2 (exceeds max. allowed)
12/18/09 01:26:27	12/18/09 01:27:46	user1@host1.ibm.com	2 (exceeds max. allowed)
12/18/09 01:29:59	12/18/09 01:30:00	user2@host1.ibm.com	1
12/18/09 01:30:00	12/18/09 01:30:00	user2@host2.ibm.com	3 (exceeds max. allowed)
12/18/09 01:30:14	12/18/09 01:30:15	user3@host1.ibm.com	3 (exceeds max. allowed)
12/18/09 01:30:14	12/18/09 01:30:14	user2@host2.ibm.com	3 (exceeds max. allowed)

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

REPORT SUMMARY

DATE: 12/18/09 TIME: 01:32:53

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

```
reporttype=maxconcurrent
maxsubdirs=0
configfile="/opt/ibmurt/1.1/config/ibmurt.cfg"
rptmaxrecords=nomax
*adjusttime=
usagefileloc="/home/testrun/ibmxlcompiler.cuf"
*sameuser=
timesort=ascend
usagefilemaxsize=nomax
usagefilemaxage=nomax
```

FILES USED:

/home/testrun/ibmxlcompiler.cuf

REPORT DETAILS

USAGE INFORMATION FOR PRODUCT: IBM XL Fortran for Linux 13.1

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

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

Exempt Users:

Dates and times where usage exceeded the maximum allowed:

Date	Time	Number of Concurrent Users	Users
------	------	----------------------------	-------

12/18/09	01:16:01	5	user3@host3.ibm.com user2@host2.ibm.com user3@host2.ibm.com user2@host1.ibm.com user1@host2.ibm.com
12/18/09	01:16:02	5	user3@host3.ibm.com user2@host2.ibm.com user3@host2.ibm.com user2@host1.ibm.com user1@host2.ibm.com
12/18/09	01:16:08	5	user3@host3.ibm.com user2@host2.ibm.com user3@host2.ibm.com user2@host1.ibm.com user1@host2.ibm.com
12/18/09	01:16:12	5	user3@host3.ibm.com user2@host2.ibm.com user3@host2.ibm.com user2@host1.ibm.com user1@host2.ibm.com
12/18/09	01:16:24	5	user3@host3.ibm.com user2@host2.ibm.com user3@host2.ibm.com user2@host1.ibm.com user1@host2.ibm.com
12/18/09	01:26:11	2	user3@host3.ibm.com user1@host1.ibm.com
12/18/09	01:26:27	2	user3@host3.ibm.com user1@host1.ibm.com
12/18/09	01:30:00	3	user2@host2.ibm.com user2@host1.ibm.com user3@host1.ibm.com
12/18/09	01:30:14	3	user2@host2.ibm.com user2@host1.ibm.com user3@host1.ibm.com
12/18/09	01:30:14	3	user2@host2.ibm.com user2@host1.ibm.com user3@host1.ibm.com

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

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

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

Pruning usage files

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

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

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

```
urt -qusagefilemaxage=30
```

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

```
urt -qusagefilemaxsize=30
```

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

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

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

- Export the file system from each machine where the usage files exist and mount it on the machine where the utilization reporting tool is installed. Then run the tool to prune the usage files on the mounted network file system.
- After copying the usage files to the machine where the utilization reporting tool is installed, delete the files and use new usage files to capture any subsequent compiler invocations. This approach might also speed up the report generation because the utilization reporting tool is not accessing the usage files remotely and it is not spending time pruning the usage files.

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

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

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

Diagnostic messages from utilization tracking and reporting

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

For example:

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

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

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

```
Unrecognized option -qmaxsubdir.
```

This message indicates that you have specified a wrong option.

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

Chapter 6. Summary of compiler options by functional category

The XL Fortran options available on the Linux platform are grouped into the following categories:

- “Output control”
- “Input control” on page 78
- “Language element control” on page 79
- “Floating-point and integer control” on page 81
- “Object code control” on page 82
- “Error checking and debugging” on page 83
- “Listings, messages, and compiler information” on page 85
- “Optimization and tuning” on page 86
- “Linking” on page 90
- “Portability and migration” on page 90
- “Compiler customization” on page 91
- “Deprecated options” on page 92

If the option supports an equivalent `@PROCESS` directive, this is indicated. To get detailed information on any option listed, see the full description page for that option.

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

In general, this document uses the convention of lowercase for `-q` compiler options and suboptions and uppercase for `@PROCESS` directives.

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.

For detailed information about each compiler option, see Chapter 7, “Detailed descriptions of the XL Fortran compiler options,” on page 95.

Output control

The options in this category control the type of file output the compiler produces, as well as the locations of the output. These are the basic options that determine the compiler components that will be invoked, the preprocessing, compilation, and linking steps that will (or will not) be taken, and the kind of output to be generated.

Table 6. Compiler output options

Option name	@PROCESS directive	Description
“-c” on page 99	None.	Prevents the completed object from being sent to the linker. With this option, the output is a .o file for each source file.

Table 6. Compiler output options (continued)

Option name	@PROCESS directive	Description
"-d" on page 100	None.	Causes preprocessed source files that are produced by <code>cpp</code> to be kept rather than deleted.
"-qmkshrobj" on page 200	None.	Creates a shared object from generated object files.
"-qmoddir" on page 201	None.	Specifies the location for any module (<code>.mod</code>) files that the compiler writes.
"-o" on page 110	None.	Specifies a name for the output object, assembler, or executable file.
"-S" on page 279	None.	Generates an assembler language file for each source file.
"-qtimestamps" on page 259	None.	Controls whether or not implicit time stamps are inserted into an object file.

Input control

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

Table 7. Compiler input options

Option name	@PROCESS directive	Description
"-D" on page 100, "-qdlines" on page 142	DLINES	Specifies whether the compiler compiles fixed source form lines with a D in column 1 or treats them as comments.
"-I" on page 104	None.	Adds a directory to the search path for include files and <code>.mod</code> files.
"-qcclines" on page 132	CCLINES	Determines whether the compiler recognizes conditional compilation lines in fixed source form and F90 free source form. This option is not supported with IBM free source form.
"-qci" on page 133	CI	Specifies the identification numbers (from 1 to 255) of the <code>INCLUDE</code> lines to process.
"-qcr" on page 134	None.	Controls how the compiler interprets the CR (carriage return) character.
"-qdirective" on page 139	DIRECTIVE	Specifies sequences of characters, known as trigger constants, that identify comment lines as compiler comment directives.

Table 7. Compiler input options (continued)

Option name	@PROCESS directive	Description
"-qfixed" on page 150	FIXED	Indicates that the input source program is in fixed source form and optionally specifies the maximum line length.
"-qfpp" on page 157	None.	Controls Fortran-specific preprocessing in the C preprocessor. This is a C preprocessor option, and must therefore be specified using the -WF option.
"-qfree" on page 159	FREE	Indicates that the source code is in free source form.
"-qmixed" on page 200, "-U" on page 281	MIXED	Makes the compiler sensitive to the case of letters in names.
"-qppsuborigarg" on page 219	None.	Instructs the C preprocessor to substitute original macro arguments before further macro expansion. This is a C preprocessor option, and must therefore be specified using the -WF option.
"-qsuffix" on page 253	None.	Specifies the source-file suffix on the command line.
"-qxlines" on page 275	XLINES	Specifies whether fixed source form lines with an X in column 1 are compiled or treated as comments.

Language element control

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

Table 8. Language element control options

Option name	@PROCESS directive	Description
"-qinit" on page 172	INIT(F90PTR)	Makes the initial association status of pointers disassociated.
"-qlanglvl" on page 187	LANGLVL	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.

Table 8. Language element control options (continued)

Option name	@PROCESS directive	Description
"-qmbcs" on page 198	MBCS	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.
"-qnullterm" on page 204	NULLTERM	Appends a null character to each character constant expression that is passed as a dummy argument, making it more convenient to pass strings to C functions.
"-1" on page 97, "-qonetrip" on page 206	ONETRIP	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.
"-qposition" on page 218	POSITION	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.
"-qqcount" on page 222	QCOUNT	Accepts the Q character-count edit descriptor (Q) as well as the extended-precision Q edit descriptor (Qw.d).
"-qsaa" on page 228	SAA	Checks for conformance to the SAA FORTRAN language definition. It identifies nonconforming source code and also options that allow such nonconformances.
"-qsave" on page 229	SAVE	Specifies the default storage class for local variables.
"-qsclk" on page 232	None.	Specifies the resolution that the SYSTEM_CLOCK intrinsic procedure uses in a program.
"-u" on page 282, "-qundef" on page 262	UNDEF	Specifies that no implicit typing of variable names is permitted. -qundef is the long form of the "-u" on page 282 option.

Table 8. Language element control options (continued)

Option name	@PROCESS directive	Description
"-qxl77" on page 268	XLF77	Provides compatibility with FORTRAN 77 aspects of language semantics and I/O data format that have changed.
"-qxl90" on page 270	XLF90	Provides compatibility with the Fortran 90 standard for certain aspects of the Fortran language.
"-qxl2003" on page 272	XLF2003	Provides the ability to use language features specific to the Fortran 2003 standard when compiling with compiler invocations that follow earlier Fortran standards, as well as the ability to disable these features when compiling with compiler invocations that follow the Fortran 2003 standard.

Floating-point and integer control

Specifying the details of how your applications perform calculations can allow you to take better advantage of your system's floating-point performance and precision, including how to direct rounding. However, keep in mind that strictly adhering to IEEE floating-point specifications can impact the performance of your application. Using the options in the following table, you can control trade-offs between floating-point performance and adherence to IEEE standards. Some of these options also allow you to control certain aspects of integer calculations.

Table 9. Floating-point and integer control options

Option name	@PROCESS directive	Description
"-qautodbl" on page 126	AUTODBL	Provides an automatic means of converting single-precision floating-point calculations to double-precision and of converting double-precision calculations to extended-precision.
"-qdpc" on page 142	DPC	Increases the precision of real constants for maximum accuracy, when assigning real constants to DOUBLE PRECISION variables.
"-qenum" on page 143	None.	Specifies the range of the enumerator constant and enables storage size to be determined.
"-qfloat" on page 152	FLOAT	Selects different strategies for speeding up or improving the accuracy of floating-point calculations.
"-qiieee" on page 170, "-y" on page 286	IEEE	Specifies the rounding mode that the compiler will use when it evaluates constant floating-point expressions at compile time.

Table 9. Floating-point and integer control options (continued)

Option name	@PROCESS directive	Description
"-qintlog" on page 178	INTLOG	Specifies that you can mix integer and logical data entities in expressions and statements.
"-qintsize" on page 179	INTSIZE	Sets the size of default INTEGER and LOGICAL data entities that have no length or kind specified.
"-qrealsize" on page 223	REALSIZE	Sets the default size of REAL , DOUBLE PRECISION , COMPLEX , and DOUBLE COMPLEX values.
"-qstrictieemod" on page 251	STRICTIEEMOD	Specifies whether the compiler will adhere to the Fortran 2003 IEEE arithmetic rules for the ieee_arithmetic and ieee_exceptions intrinsic modules.

Object code control

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

Table 10. Object code control options

Option name	@PROCESS directive	Description
"-q32" on page 112	None.	Enables 32-bit compilation mode (or, more briefly, 32-bit mode) support in a 64-bit environment.
"-q64" on page 113	None.	Indicates 64-bit compilation bit mode and, together with the -qarch option, determines the target machines on which the 64-bit executable will run.
"-qinlglue" on page 175	INLGLUE	When used with -O2 or higher optimization, inlines glue code that optimizes external function calls in your application.
"-qpic" on page 215	None.	Generates Position-Independent Code suitable for use in shared libraries.
"-qsaveopt" on page 230	None.	Saves the command-line options used for compiling a source file, the user's configuration file name and the options specified in the configuration files, the version and level of each compiler component invoked during compilation, and other information to the corresponding object file.

Table 10. Object code control options (continued)

Option name	@PROCESS directive	Description
"-qstackprotect" on page 244	None.	Provides protection against malicious code or programming errors that overwrite or corrupt the stack.
"-qtbtable" on page 257	None.	Controls the amount of debugging traceback information that is included in the object files.
"-qthreaded" on page 258	None.	Indicates to the compiler whether it must generate threadsafe code.

Error checking and debugging

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

To control the amount and type of information you receive regarding the behavior and performance of your application, consult "Listings, messages, and compiler information" on page 85.

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

Table 11. Error checking and debugging options

Option name	@PROCESS directive	Description
"-#" on page 96	None.	Previews the compilation steps specified on the command line, without actually invoking any compiler components.
"-C" on page 98, "-qcheck" on page 132	CHECK	Checks each reference to an array element, array section, or character substring to ensure the reference stays within the defined bounds of the entity.
"-g" on page 103, "-qdbg" on page 136	DBG	Generates debug information for use by a symbolic debugger.
"-qflttrap" on page 158	FLTTRAP	Determines what types of floating-point exception conditions to detect at run time.

Table 11. Error checking and debugging options (continued)

Option name	@PROCESS directive	Description
"-qfullpath" on page 161	FULLPATH	When used with the -g or -qlinedebug option, this option records the full, or absolute, path names of source and include files in object files compiled with debugging information, so that debugging tools can correctly locate the source files.
"-qfunctrace" on page 162	None.	Traces entry and exit points of procedures in your program. If your program contains C++ compilation units, this option also traces C++ catch blocks.
"-qfunctrace_xlf_catch" on page 163	None.	Specifies the name of the catch tracing subroutine.
"-qfunctrace_xlf_enter" on page 164	None.	Specifies the name of the entry tracing subroutine.
"-qfunctrace_xlf_exit" on page 165	None.	Specifies the name of the exit tracing subroutine.
"-qhalt" on page 166	HALT	Stops compilation before producing any object, executable, or assembler source files if the maximum severity of compile-time messages equals or exceeds the severity you specify.
"-qinfo" on page 171	None.	Produces or suppresses groups of informational messages.
"-qinitauto" on page 172	INITAUTO	Initializes uninitialized automatic variables to a specific value, for debugging purposes.
"-qkeepparm" on page 187	None.	When used with -O2 or higher optimization, specifies whether function parameters are stored on the stack.
"-qlinedebug" on page 191	None.	Generates only line number and source file name information for a debugger.
"-qobject" on page 205	OBJECT	Specifies whether to produce an object file or to stop immediately after checking the syntax of the source files.
"-qoptdebug" on page 206	None.	When used with high levels of optimization, produces files containing optimized pseudocode that can be read by a debugger.

Table 11. Error checking and debugging options (continued)

Option name	@PROCESS directive	Description
"-qsigtrap" on page 234	None.	Sets up the specified trap handler to catch SIGTRAP and SIGFPE exceptions when compiling a file that contains a main program.
"-qwarn64" on page 266	None.	Displays informational messages identifying statements that may cause problems with 32-bit to 64-bit migration.
"-qxflag=dvz" on page 266	None.	Causes the compiler to generate code to detect floating-point divide-by-zero operations.

Listings, messages, and compiler information

The options in the following table allow you control over the listing file, as well as how and when to display compiler messages. You can use these options in conjunction with those described in "Error checking and debugging" on page 83 to provide a more robust overview of your application when checking for errors and unexpected behavior.

Table 12. Listings and messages options

Option name	@PROCESS directive	Description
"-qattr" on page 125	ATTR	Produces a compiler listing that includes the attribute component of the attribute and cross-reference section of the listing.
"-qflag" on page 151	FLAG	Limits the diagnostic messages to those of a specified severity level or higher.
"-qlist" on page 192	LIST	Produces a compiler listing file that includes an object listing.
"-qlistfmt" on page 193	None.	Creates an XML report to assist with finding optimization opportunities.
"-qlistopt" on page 195	None.	Produces a compiler listing file that includes all options in effect at the time of compiler invocation.
"-qphsinfo" on page 213	PHSINFO	Reports the time taken in each compilation phase to standard output.

Table 12. Listings and messages options (continued)

Option name	@PROCESS directive	Description
"-qnoprint" on page 203	None.	Prevents the compiler from creating the listing file, regardless of the settings of other listing options.
"-qreport" on page 226	None.	Produces listing files that show how sections of code have been optimized.
"-qsource" on page 242	SOURCE	Produces a compiler listing file that includes the source section of the listing and provides additional source information when printing error messages.
"-qsuppress" on page 254	None.	Prevents specific informational or warning messages from being displayed or added to the listing file, if one is generated.
"-qversion" on page 264	None.	Displays the version and release of the compiler being invoked.
"-V" on page 283	None.	The same as -v except that you can cut and paste directly from the display to create a command.
"-v" on page 283	None.	Reports the progress of compilation, by naming the programs being invoked and the options being specified to each program.
"-w" on page 286	None.	Suppresses informational, language-level and warning messages (equivalent to -qflag=e:e).
"-qxref" on page 277	XREF	Produces a compiler listing that includes the cross-reference component of the attribute and cross-reference section of the listing.

Optimization and tuning

You can control the optimization and tuning process, which can improve the performance of your application at run time, using the options in the following table. Remember that not all options benefit all applications. Trade-offs sometimes occur between an increase in compile time, a reduction in debugging capability, and the improvements that optimization can provide. In addition to the option

descriptions in this section, consult the *XL Fortran Optimization and Programming Guide* for details on the optimization and tuning process as well as writing optimization friendly source code.

Some of the options in “Floating-point and integer control” on page 81 can also improve performance, but you must use them with care to ensure your application retains the floating point semantics it requires.

Table 13. Optimization and tuning options

Option name	@PROCESS directive	Description
“-qalias” on page 114	ALIAS(<i>argument_list</i>)	Indicates whether a program contains certain categories of aliasing or does not conform to Fortran standard aliasing rules. The compiler limits the scope of some optimizations when there is a possibility that different names are aliases for the same storage location..
“-qarch” on page 120	None.	Specifies the processor architecture, or family of architectures, where the code may run. This allows the compiler to take maximum advantage of the machine instructions specific to an architecture, or common to a family of architectures.
“-qassert” on page 124	ASSERT	Provides information about the characteristics of your code that can help the compiler fine-tune optimizations..
“-qcache” on page 129	None.	When specified with -O4 , -O5 , or -qipa , specifies the cache configuration for a specific execution machine.
“-qcompact” on page 134	COMPACT	Avoids optimizations that increase code size.
“-qdirectstorage” on page 141	None.	Informs the compiler that a given compilation unit may reference write-through-enabled or cache-inhibited storage.
“-qsimd” on page 234	None.	Controls whether the compiler can automatically take advantage of vector instructions for processors that support them.
“-qessl” on page 146	None.	Allows the compiler to substitute the Engineering and Scientific Subroutine Library (ESSL) routines in place of Fortran 90 intrinsic procedures.

Table 13. Optimization and tuning options (continued)

Option name	@PROCESS directive	Description
"-qhot" on page 167	HOT(<i>suboptions</i>)	Performs high-order loop analysis and transformations (HOT) during optimization.
-qinline	None.	Attempts to inline procedures instead of generating calls to those procedures, for improved performance.
"-qipa" on page 181	None.	Enables or customizes a class of optimizations known as interprocedural analysis (IPA).
"-qlibansi" on page 189	None.	Assumes that all functions with the name of an ANSI C library function are, in fact, the library functions and not a user function with different semantics.
-qlibmpi	None.	Asserts that all functions with Message Passing Interface (MPI) names are in fact MPI functions and not a user function with different semantics.
"-qmaxmem" on page 197	MAXMEM	Limits the amount of memory that the compiler allocates while performing specific, memory-intensive optimizations to the specified number of kilobytes.
"-qminimaltoc" on page 199	None.	In 64-bit compilation mode, minimizes the number of entries in the global entity table of contents (TOC).
"-O" on page 108	OPTIMIZE	Specifies whether to optimize code during compilation and, if so, at which level.
"-p" on page 111	None.	Prepares the object files produced by the compiler for profiling.
"-qpdf1, -qpdf2" on page 208	None.	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.
"-qprefetch" on page 220	None.	Inserts prefetch instructions automatically where there are opportunities to improve code performance.

Table 13. Optimization and tuning options (continued)

Option name	@PROCESS directive	Description
"-qshowpdf" on page 233	None.	When used with -qpdf1 and a minimum optimization level of -O2 at compile and link steps, inserts additional profiling information into the compiled application to collect call and block counts for all procedures in the application.
"-qsmallstack" on page 236	None.	Minimizes stack usage where possible.
"-qsmp" on page 237	None.	Enables parallelization of program code.
"-qstacktemp" on page 245	None.	Determines where to allocate certain XL Fortran compiler temporaries at run time.
"-qstrict" on page 247	STRICT	Ensures that optimizations done by default at optimization levels -O3 and higher, and, optionally at -O2 , do not alter certain program semantics mostly related to strict IEEE floating-point conformance.
"-qstrict_induction" on page 252	None.	Prevents the compiler from performing induction (loop counter) variable optimizations. These optimizations may be unsafe (may alter the semantics of your program) when there are integer overflow operations involving the induction variables.
"-qtune" on page 260	TUNE	Tunes instruction selection, scheduling, and other architecture-dependent performance enhancements to run best on a specific hardware architecture.
"-qunroll" on page 262	UNROLL	Specifies whether unrolling DO loops is allowed in a program. Unrolling is allowed on outer and inner DO loops.
"-qunwind" on page 264	None.	Specifies that the compiler will preserve the default behavior for saves and restores to volatile registers during a procedure call.
"-qzerosize" on page 278	None.	Determines whether checking for zero-sized character strings and arrays takes place in programs that might process such objects.

Linking

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

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

Table 14. Linking options

Option name	@PROCESS directive	Description
"-e" on page 101	None.	When used together with the -qmkshrobj , specifies an entry point for a shared object.
"-L" on page 105	None.	At link time, searches the directory path for library files specified by the -l option.
"-l" on page 106	None.	Searches for the specified library file, <i>libkey.so</i> , and then <i>libkey.a</i> for dynamic linking, or just for <i>libkey.a</i> for static linking.
"-qstaticlink" on page 246	None.	Controls how shared and nonshared runtime libraries are linked into an application.

Portability and migration

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

Table 15. Portability and migration options

Option name	@PROCESS directive	Description
"-qalign" on page 118	ALIGN	Specifies the alignment of data objects in storage, which avoids performance problems with misaligned data.
"-qbindcextname" on page 128	BINDCEXTNAME	Controls whether the -qextname option affects BIND(C) entities.
"-qctypless" on page 135	CTYPLSS	Specifies whether character constant expressions are allowed wherever typeless constants may be used.
"-qddim" on page 137	DDIM	Specifies that the bounds of pointer arrays are re-evaluated each time the arrays are referenced and removes some restrictions on the bounds expressions for pointer arrays.

Table 15. Portability and migration options (continued)

Option name	@PROCESS directive	Description
"-qdescriptor" on page 138	None.	Specifies the XL Fortran internal descriptor data structure format to use for non object-oriented entities in your compiled applications.
"-qescape" on page 144	ESCAPE	Specifies how the backslash is treated in character strings, Hollerith constants, H edit descriptors, and character string edit descriptors.
"-qextern" on page 147	EXTERN	Allows user-written procedures to be called instead of XL Fortran intrinsics.
"-qextname" on page 148	EXTNAME	Adds an underscore to the names of all global entities.
"-qmodule" on page 202	None.	Specifies that the compiler should use the XL Fortran Version 8.1 naming convention for non-intrinsic module files.
"-qport" on page 216	PORT	Provides options to accommodate other Fortran language extensions when porting programs to XL Fortran.
"-qswapomp" on page 256	SWAPOMP	Specifies that the compiler should recognize and substitute OpenMP routines in XL Fortran programs.

Compiler customization

The options in the following table allow you to specify alternate locations for compiler components, configuration files, standard include directories, and internal compiler operation. You should only need to use these options in specialized installation or testing scenarios.

Table 16. Compiler customization options

Option name	@PROCESS directive	Description
"-B" on page 97	None.	Determines substitute path names for XL Fortran executables such as the compiler, assembler, linker, and preprocessor.
"-F" on page 102	None.	Specifies an alternative configuration file, which stanza to use within the configuration file, or both.
"-NS" on page 107, "-qspillsize" on page 243	SPILLSIZE	Specifies the size (in bytes) of the register spill space; the internal program storage areas used by the optimizer for register spills to storage.

Table 16. Compiler customization options (continued)

Option name	@PROCESS directive	Description
"-qalias_size" on page 117	ALIAS_SIZE(<i>bytes</i>)	Specifies an appropriate initial size, in bytes, for the aliasing table. This option has effect only when optimization is enabled.
"-t" on page 280	None.	Applies the prefix specified by the -B option to the designated components.
"-W" on page 284	None.	Passes the listed options to a component that is executed during compilation.

Deprecated options

The compiler still accepts options listed in the following table. Those options that the compiler supports under another name may not perform as previously documented.

An option is 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.

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.

Table 17. Deprecated options

Option name	Replacement option
-Q	This option is no longer supported. It is replaced by <code>-qinline</code> .
-qcharlen= <i>length</i>	Obsolete. It is still accepted, but it has no effect. The maximum length for character constants and subobjects of constants is 32 767 bytes (32 KB). The maximum length for character variables is 268 435 456 bytes (256 MB) in 32-bit mode. The maximum length for character variables is 2**40 bytes in 64-bit mode. These limits are always in effect and are intended to be high enough to avoid portability problems with programs that contain long strings
-qarch=com	This suboption is no longer supported. Replaced by <code>-qarch=ppc64grsq</code> .
-qbigdata	Obsolete. This option is no longer supported.
-qenablevmx	Replaced by <code>-qsimd=auto</code> .
-qhot=simd <u>nosimd</u>	Obsolete. Replaced by " <code>-qsimd</code> " on page 234.
-qipa=clonearch <u>noclonearch</u>	replaced by <code>-qtune=blanced</code> .

Table 17. *Deprecated options (continued)*

Option name	Replacement option
-qipa=cloneproc <u>nocloneproc</u>	replaced by -qtune=blanced.
-qipa=inline <u>noinline</u>	This option and all of its suboptions, including auto , noauto , limit , and threshold are no longer supported. This option is replaced by -qinline.
-qipa=pdfname	Replaced by -qpdf1=pdfname, -qpdf2=pdfname.
-qposition=append	-qposition=appendunknown replaces the -qposition=append suboption.
-qrecur -qnorecur	<p>Not recommended. Specifies whether external subprograms may be called recursively.</p> <p>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.</p>

Chapter 7. Detailed descriptions of the XL Fortran compiler options

This section contains descriptions of the individual options available in XL Fortran.

For each option, the following information is provided:

Category

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

@PROCESS

For many compiler options, you can use an equivalent @PROCESS directive to apply the option's functionality within the source code, limiting the scope of the option's application to a single source file or compilation unit, or even selected sections of code.

Purpose

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

Syntax

This section provides the syntax for the command-line option and for the equivalent @PROCESS directive, if applicable. Syntax is shown first in command-line form, and then in @PROCESS form. For an explanation of the notations used to represent command-line syntax, see "Conventions" on page viii.

Uppercase letters are sometimes used to indicate the minimum number of characters for an option. For example, in **-qassert=CONTIGuous**, the uppercase letters **CONTIG** indicate the minimum number of characters you must use for this option. Therefore if you use **-qassert=contig** or **-qassert=contigu**, the compiler recognizes both as valid.

For @PROCESS syntax, the following notations are used:

- 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 (...).

Defaults

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

Parameters

This section describes the suboptions that are available for the option.

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

Examples

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

-#

Category

Error checking and debugging

@PROCESS

None.

Purpose

Previews the compilation steps specified on the command line, without actually invoking any compiler components.

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

Syntax

Option:

►► -# ◄◄

Defaults

Not applicable.

Usage

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. (If you are familiar with the `make` command, it is similar to `make -n`.)

This option produces the same output as `-v` and `-V`, but does not perform the compilation.

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

- “-v” on page 283
- “-V” on page 283

-1

Category

Language element control

Purpose

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.

`-qonetrip` is the long form of `-1`.

Syntax

Option:

►► `-1` _____ ◀◀

@PROCESS:

@PROCESS ONETRIP | NOONETRIP

Defaults

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.

-B

Category

Compiler customization

Purpose

Determines substitute path names for XL Fortran executables such as the compiler, assembler, linker, and preprocessor.

It can be used in combination with the `-t` option, which determines which of these components are affected by `-B`.

Syntax

►► `-B` prefix _____ ◀◀

@PROCESS:

@PROCESS `-Bprefix`

Defaults

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

Parameters

prefix

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

Usage

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.

Related information

- “-t” on page 280
- “-F” on page 102
- “Using custom compiler configuration files” on page 10
- “Running two levels of XL Fortran” on page 19

-C

Category

Error checking and debugging

Purpose

Checks each reference to an array element, array section, or character substring to ensure the reference stays within the defined bounds of the entity.

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

Syntax

Option:

▶▶ -C ◀◀

@PROCESS:

@PROCESS CHECK | NOCHECK

Defaults

-qnocheck

Usage

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

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

Because runtime 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).

The **-C** option prevents some optimizations. You may want to remove the **-C** option after the debugging of your code is complete and then add any desired optimization options for better performance.

The valid bounds for character substring expressions differ depending on the setting of the **-qzerosize** option.

Related information

- “-qhot” on page 167
- “-qzerosize” on page 278
- “-qsigtrap” on page 234 and *Installing an exception handler* in the *XL Fortran Optimization and Programming Guide* describe how to detect and recover from **SIGTRAP** signals without ending the program.

-C

Category

Object code control

@PROCESS

None.

Purpose

Prevents the completed object from being sent to the linker. With this option, the output is a .o file for each source file.

Syntax

►► -C ◀◀

Defaults

Not applicable.

Usage

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

- “-o” on page 110.

-D

Category

Input control

Purpose

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

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

Syntax

▶— -D —▶

@PROCESS:

@PROCESS `DLINES` | `NODLINES`

Usage

If you specify `-D`, 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.

Note that in order to pass C-style `-D` macro definitions to the C preprocessor, for example, when compiling a file that ends with `.F`, use the `-W` option. For example:
`-WF, -DDEFINE_THIS`

-d

Category

Output control

@PROCESS

None.

Purpose

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

Syntax

▶▶ — *-d* —————▶▶▶▶

Defaults

Not applicable.

Results

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

Related information

- “Passing Fortran files through the C preprocessor” on page 31

-e

Category

Linking

@PROCESS

None.

Purpose

When used together with the **-qmkshrobj**, specifies an entry point for a shared object.

Syntax

▶▶ — *-e* — *entry_name* —————▶▶▶▶

Defaults

None.

Parameters

name

The name of the entry point for the shared executable.

Usage

Specify the **-e** option only with the **-qmkshrobj** option. For more information, see the description for **-qmkshrobj**.

Note: When you link object files, do not use the `-e` option. The default entry point of the executable output is `__start`. Changing this label with the `-e` flag can produce errors.

Related information

- “-qmkshrobj” on page 200

-F

Category

Compiler customization

@PROCESS

None.

Purpose

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.

Syntax

```
► -F config_file [ :—stanza ]
```

Defaults

By default, the compiler uses the configuration file that is configured at installation time, and the stanza defined in that file for the invocation command currently being used (for example, `xlF2003`, `xlF90_r`, `xlF90`, and so on.).

Parameters

config_file

The full path name of the alternate compiler configuration file to use.

stanza

The name of the configuration file stanza to use for compilation. This directs the compiler to use the entries under that *stanza* regardless of the invocation command being used. For example, if you are compiling with `xlF95`, but you specify the `xlF95` stanza, the compiler will use all the settings specified in the `xlF95` stanza.

Usage

A simple way to customize the way the compiler works, as an alternative to writing complicated compilation scripts, is to add new stanzas to `/opt/ibmcomp/xlf/13.1/etc/xlf.cfg`, giving each stanza a different name and a different set of default compiler options. Or, you can specify a user-defined

configuration file with the `XL_F_USR_CONFIG` environment variable rather than editing the default configuration file. You may find a 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. Note that the settings in any user-defined configuration file are processed before the ones specified by the `-F` option.

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.

Alternatively, you can store customized settings in the user-defined configuration file specified by the `XL_F_USR_CONFIG` environment variable. This file will not be replaced during reinstallation.

Examples

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

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

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

Related information

- “Creating custom configuration files” on page 11 explains the contents of a custom, user-defined configuration file and shows how to select different stanzas in the file without using the `-F` option.
- “Editing the default configuration file” on page 14 explains how to edit the contents of a configuration file for use with the `-F` option.
- “-B” on page 97
- “-t” on page 280
- “-W” on page 284

-g

Category

Error checking and debugging

Purpose

Generates debug information for use by a symbolic debugger.

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

`-g` implies the `-qnoinline` option.

Syntax

Option:

►► `-g` ◀◀

@PROCESS:

@PROCESS `DBG` | `NODBG`

Related information

- `-qinline`
- “`-qlinedebg`” on page 191
- “Debugging a Fortran program” on page 299
- “Symbolic debugger support” on page 6

-I

Category

Input control

@PROCESS

None.

Purpose

Adds a directory to the search path for include files and `.mod` files.

Syntax

►► `-I`*path_name* ◀◀

Defaults

Not applicable.

Parameters

path_name

A valid path name (for example, `/home/dir`, `/tmp`, or `./subdir`).

Usage

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.

Rules

The compiler appends a / to *path_name* 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 *path_name* 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/ibmcomp/xlf/13.1/include where include and .mod files shipped with the compiler are located.

Related information

- “-qmoddir” on page 201
- “-qfullpath” on page 161

-k

Category

Input control

Purpose

Indicates that the source code is in free source form.

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

Syntax

Option:

▶▶ -k ◀◀

@PROCESS:

@PROCESS FREE(F90)

Related information

- “-qfree” on page 159
- *Free source form* in the *XL Fortran Language Reference*.

-L

Category

Linking

@PROCESS

None.

Purpose

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

Syntax

Option:

►► — *-l—Directory* —————►►

Defaults

Not applicable.

Usage

Adds *Directory* to the list of search directories that are used for finding libraries designated by the **-l** flag (lowercase letter **L**). If you use libraries other than the default ones specified in `/opt/ibmcmp/xlf/13.1/lib` or `/opt/ibmcmp/xlf/13.1/lib64`, 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

- “Linking” on page 90
- “Linking XL Fortran programs” on page 33

-l

Category

Linking

@PROCESS

None.

Purpose

Searches for the specified library file, *libkey.so*, and then *libkey.a* for dynamic linking, or just for *libkey.a* for static linking.

Syntax

►► — *-l—key* —————►►

Defaults

The compiler default is to search only for some of the compiler runtime libraries. The default configuration file specifies the default library names to search for with

the `-l` compiler option, and the default search path for libraries with the `-L` compiler option.

Parameters

key

The name of the library minus the `lib` prefix.

Rules

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

Related information

- “Linking” on page 90
- “Linking XL Fortran programs” on page 33

-NS

Category

Compiler customization

Purpose

Specifies the size (in bytes) of the register spill space; the internal program storage areas used by the optimizer for register spills to storage.

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

Syntax

Option:

▶▶ `-NS—bytes` —————▶▶

@PROCESS:

@PROCESS SPILLSIZE(*bytes*)

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.

Parameters

bytes

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.

Related information

- “`-qspillsize`” on page 243

-O

Category

Optimization and tuning

Purpose

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

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

Syntax

Option:



@PROCESS:

@PROCESS OPTimize[*level*] | NOOPTimize

Defaults

`nooptimize` or `-O0` or `optimize=0`

Parameters

not specified

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

- O** For each release of XL Fortran, `-O` enables the level of optimization that represents the best tradeoff between compilation speed and runtime 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 is ignored and has no effect on the outcome of the compilation.
- 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 runtime 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**.

Specifying **-O3** implies **-qhot=level=0**, unless you explicitly specify **-qhot** or **-qhot=level=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. For example, specifying **-O4 -qarch=ppc** allows aggressive intraprocedural optimization while maintaining code portability.

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

Note:

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

Usage

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 runtime 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 maintains the same program behavior as with **-O2**, at the cost of optimization opportunity. Refer to the **-qstrict** option for the list of optimizations it disables.

If the **-O** option is used in an **@PROCESS** statement, only an optimization level of 0, 2, or 3 is allowed. Note that unlike using **-O3** in command line invocation, specifying **@PROCESS OPT(3)** does not imply **-qhot=level=0**.

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” on page 247 shows how to turn off the effects of **-O3** that might change the semantics of a program.
- “-qipa” on page 181, “-qhot” on page 167, and “-qpdf1, -qpdf2” on page 208 turn on additional optimizations that may improve performance for some programs.
- “Optimizing your applications” in the *XL Fortran Optimization and Programming Guide* discusses technical details of the optimization techniques the compiler uses and some strategies you can use to get maximum performance from your code.

-o

Category

Output control

@PROCESS

None.

Purpose

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

Syntax

►► `-o—name` ◀◀

Defaults

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

Usage

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

Rules

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

Examples

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

-p

Category

Optimization and tuning

@PROCESS


None.

Purpose

Prepares the object files produced by the compiler for profiling.

The compiler produces monitoring code that counts the number of times each routine is called. The compiler inserts a call to the monitor subroutine at the start of each subprogram.

Syntax

►► -p 

Defaults

Not applicable.

Usage

When you run a program compiled with **-p** or **-pg** and it ends normally, it produces a **gmon.out** file with the profiling information. You can then use the **gprof** command to generate a runtime profile.

Examples

```
$ xlf95 -pg needs_tuning.f
$ a.out
$ gprof
.
.
.
detailed and verbose profiling data
.
.
.
```

Related information

- Refer to your operating system documentation for more information on profiling and the **gprof** command.

-q32

Category

Object code control

@PROCESS

None.

Purpose

Enables 32-bit compilation mode (or, more briefly, 32-bit mode) support in a 64-bit environment.

Syntax

►► — -q—32 —————►►

Defaults

-q32 is the default.

Usage

- The default integer and default real size are 4 bytes in 32-bit mode.
- The default integer pointer size is 4 bytes in 32-bit mode.
- 32-bit object modules are created when targeting 32-bit mode.
- **-q64** will override **-q32**.
- All settings for **-qarch** are compatible with **-q32**. If you specify **-q32**, the default suboption is **ppc64grsq**, and the default **-qtune** suboption for **-q32** is **pwr4**.
- The **LOC** intrinsic returns an **INTEGER(4)** value.

Examples

- Using 32-bit compilation mode and targeting a generic PowerPC architecture:
-qarch=ppc -q32
- Now keep the same compilation mode, but change the target to POWER5:
-qarch=ppc -q32 -qarch=pwr5
Notice that the last setting for **-qarch** wins.
- Now keep the same target, but change the compilation mode to 64-bit:
-qarch=ppc -q32 -qarch=pwr5 -q64

Notice that specifying **-q64** overrides the earlier instance of **-q32**.

Related information

- “-q64” on page 113
- “-qarch” on page 120
- “-qtune” on page 260
- “-qwarn64” on page 266
- Chapter 8, “Using XL Fortran in a 64-bit environment,” on page 289

-q64

Category

Object code control

@PROCESS

None.

Purpose

Indicates 64-bit compilation bit mode and, together with the **-qarch** option, determines the target machines on which the 64-bit executable will run.

The object module will be created in 64-bit object format and that the 64-bit instruction set will be generated. Note that you may compile in a 32-bit environment to create 64-bit objects, but you must link them in a 64-bit environment with the **-q64** option.

Syntax

►► -q64 ◀◀

Purpose

Defaults

Not applicable.

Rules

- Settings for **-qarch** that are compatible with **-q64** are as follows:
 - **-qarch=auto** (if compiling on a 64-bit system)
 - **-qarch=ppc** (With **-q64** and **-qarch=ppc**, the compiler will silently upgrade the arch to **ppc64grsq**.)
 - **-qarch=ppcgr** (With **-q64** and **-qarch=ppcgr**, the compiler will silently upgrade the arch to **ppc64grsq**.)
 - **-qarch=ppc64**
 - **-qarch=ppc64v**
 - **-qarch=ppc64gr**
 - **-qarch=ppc64grsq**
 - **-qarch=rs64a**
 - **-qarch=rs64b**
 - **-qarch=rs64c**
 - **-qarch=pwr3**
 - **-qarch=pwr4**
 - **-qarch=pwr5**
 - **-qarch=pwr5x**
 - **-qarch=pwr6**
 - **-qarch=pwr6e**
 - **-qarch=ppc970**
- The default **-qarch** setting for **-q64** is **ppc64**.
- 64-bit object modules are created when targeting 64-bit mode.
- **-q32** may override **-q64**.

- **-q64** will override a conflicting setting for **-qarch** and will result in the setting **-q64 -qarch=ppc64** along with a warning message.
- The default tune setting for **-q64** is **-qtune=pwr4**.
- The default integer and default real size is 4 bytes in 64-bit mode.
- The default integer pointer size is 8 bytes in 64-bit mode.
- The maximum array size increases to approximately 2^{40} bytes (in static storage) or 2^{60} bytes (in dynamic allocation on the heap). The maximum dimension bound range is extended to -2^{63} , $2^{63}-1$. The theoretical maximum array size is 2^{60} bytes, but this is subject to the limitations imposed by the operating system. The maximum array size for array constants has not been extended and will remain the same as the maximum in 32-bit mode. The maximum array size that you can initialize is 2^{28} bytes.
- The maximum iteration count for array constructor implied DO loops increases to $2^{63}-1$.
- The maximum character variable length extends to approximately 2^{40} bytes. The maximum length of character constants and subobjects of constants remains the same as in 32-bit mode, which is 32 767 bytes (32 KB).
- The **LOC** intrinsic returns an **INTEGER(8)** value.
- If you must use **-qautodbl=dblpad** in 64-bit mode, you should use **-qintsize=8** to promote **INTEGER(4)** to **INTEGER(8)** for 8 byte integer arithmetic.

Examples

This example targets the POWER5 in 64-bit mode:

```
-q32 -qarch=pwr5 -q64
```

This 64-bit compilation example targets the common group of 64-bit architectures:

```
-q64 -qarch=ppc
```

The arch setting is silently upgraded to **ppc64grsq**, the most "common" 64-bit mode compilation target.

Related information

- “-qarch” on page 120
- “-qtune” on page 260
- Chapter 8, “Using XL Fortran in a 64-bit environment,” on page 289
- “-qwarn64” on page 266

-qalias

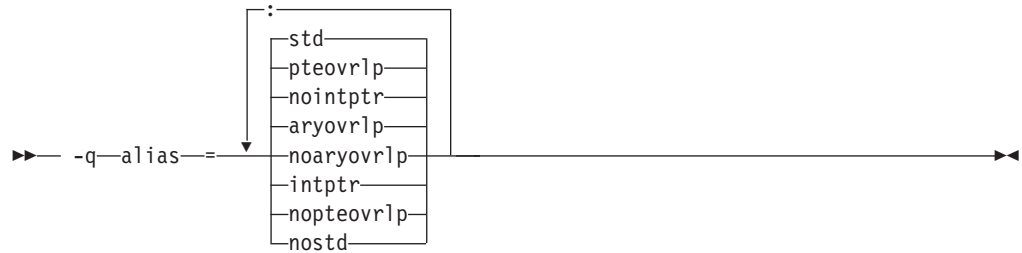
Category

Optimization and tuning

Purpose

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

Syntax



@PROCESS:

`@PROCESS ALIAS({ARGUMENT_LIST})`

Defaults

`-qalias=aryovrlp:nointptr:pteovrlp:std`

Parameters

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**.

Usage

An alias exists when an item in storage can be referred to by more than one name. The Fortran 90, Fortran 95, and Fortran 2003 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 in some other way than through the dummy argument.
- A subscript to an array within a common block exceeds that array's bounds.

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 **xlfr_r**, **xlfr**, and **f77/fort77** commands assume that integer **POINTERS** may be present (**-qalias=aryovrlp:pteovrlp:std:intptr**), while all other invocation commands assume that a program contains only standard aliasing (**-qalias=aryovrlp:pteovrlp:std:nointptr**).

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

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

```

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)

```

Related information

- See *Optimizing your applications* in the *XL Fortran Optimization and Programming Guide* for information on aliasing strategies you should consider.

-qalias_size

Category

Compiler customization

Purpose

Specifies an appropriate initial size, in bytes, for the aliasing table. This option has effect only when optimization is enabled.

Syntax

▶▶ -qalias_size=*size* ▶▶

@PROCESS:

@PROCESS ALIAS_SIZE(*size*)

Defaults

None.

Parameters

size

The initial size of the alias table, in bytes.

Usage

Compiling very large programs with optimization can cause aliasing tables to get very large, which may result in memory fragmentation. Use this option only when the compiler issues an error message with a suggested value for *size*. Specifying this option in other situations, or with values not recommended by the compiler, may cause the compiler to run out of memory.

-qalign

Category

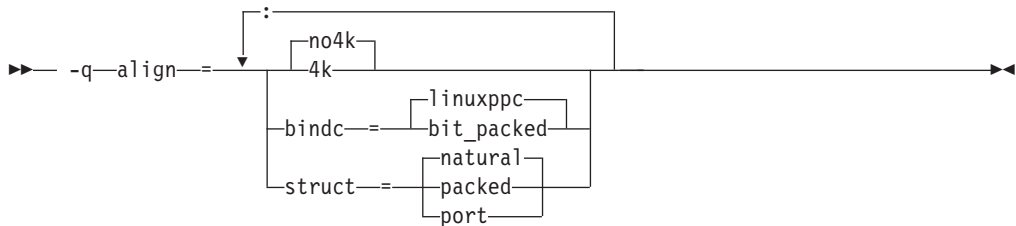
Portability and migration

Purpose

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

The **[no]4k**, **bindc**, and **struct** options can be specified and are not mutually exclusive. The **[no]4k** option is useful primarily in combination with logical volume I/O and disk striping.

Format



@PROCESS:

```
@PROCESS ALIGN({[NO]4K|STRUCT{(suboption)}|BINDC{(suboption)}})
```

Defaults

-qalign= no4k:struct=natural:bindc=linuxppc.

Parameters

[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.

bindc={suboption}

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

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

struct={suboption}

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

packed

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

natural

If the **natural** suboption of the **struct** option is specified, objects of a derived type are stored with sufficient padding such 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.

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 “-qattr” on page 125. This listing also shows the offsets of data objects.

-qarch

Category

Optimization and tuning

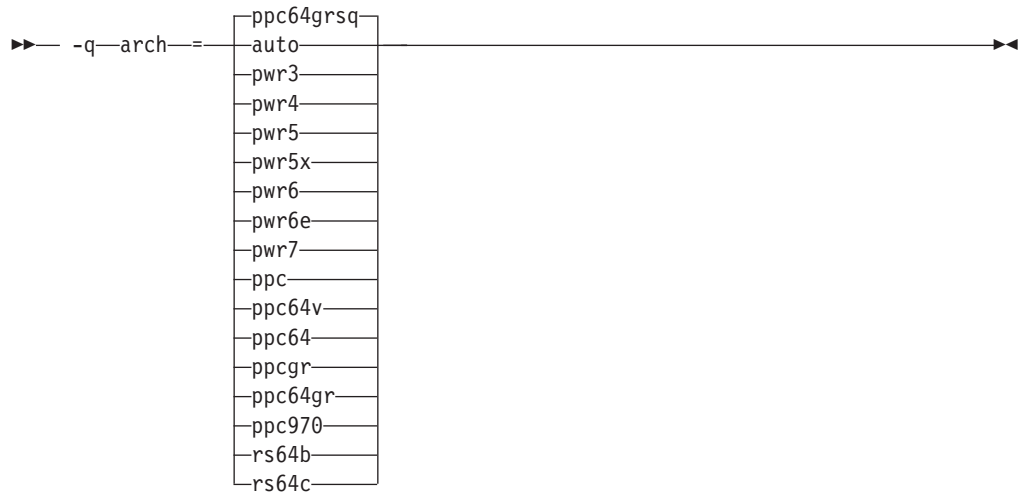
@PROCESS

None.

Purpose

Specifies the processor architecture, or family of architectures, where the code may run. This allows the compiler to take maximum advantage of the machine instructions specific to an architecture, or common to a family of architectures.

Syntax



Defaults

- `-qarch=ppc64grsq`
- `-qarch=auto` when `-O4` or `-O5` is in effect.

Parameters

auto

Automatically detects the specific architecture of the compiling machine. It assumes that the execution environment will be the same as the compilation environment. This option is implied if the `-O4` or `-O5` option is set or implied.

pwr3

Produces object code containing instructions that will run on the POWER3, POWER4, POWER5, POWER5+, POWER6, POWER7, or PowerPC 970 hardware platforms.

pwr4

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

pwr5

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

pwr5x

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

pwr6

Produces object code containing instructions that will run on the POWER6 or POWER7 hardware platforms running in POWER6 or POWER7 architected mode.

pwr6e

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

pwr7

Produces object code containing instructions that will run on the POWER7 hardware platforms.

ppc

In 32-bit mode, produces object code containing instructions that will run on any of the 32-bit PowerPC hardware platforms. This suboption causes the compiler to produce single-precision instructions to be used with single-precision data. Specifying **-qarch=ppc** together with **-q64** silently upgrades the architecture setting to **-qarch=ppc64grsq**.

ppc64

Produces object code that will run on any of the 64-bit PowerPC hardware platforms. This suboption can be selected when compiling in 32-bit mode, but the resulting object code may include instructions that are not recognized or behave differently when run on 32-bit PowerPC platforms.

ppcgr

In 32-bit mode, produces object code for PowerPC processors that support optional graphics instructions. Specifying **-qarch=ppcgr** together with **-q64** silently upgrades the architecture setting to **-qarch=ppc64grsq**.

ppc64gr

Produces code for any 64-bit PowerPC hardware platform that supports optional graphics instructions.

ppc64grsq

Produces code for any 64-bit PowerPC hardware platform that supports optional graphics and square root instructions.

ppc64v

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

ppc970

Generates instructions specific to the PowerPC 970 architecture.

rs64b

Produces object code that will run on RS64II platforms.

rs64c

Produces object code that will run on RS64III platforms.

Usage

All PowerPC machines share a common set of instructions, but may also include additional instructions unique to a given processor or processor family. Using the **-qarch** option to target a specific architecture for the compilation results in code that may not run on other architectures, but provides the best performance for the selected architecture. If you want maximum performance on a specific architecture and will not be using the program on other architectures, use the appropriate architecture option. If you want to generate code that can run on more than one architecture, specify a **-qarch** suboption that supports a group of architectures. Table 18 shows the features supported by the different processor architectures and their representative **-qarch** suboptions:

Table 18. Feature support in processor architectures

Architecture	Graphics support	Square root support	64-bit support	Vector processing support
rs64b	yes	yes	yes	no

Table 18. Feature support in processor architectures (continued)

Architecture	Graphics support	Square root support	64-bit support	Vector processing support
rs64c	yes	yes	yes	no
pwr3	yes	yes	yes	no
pwr4	yes	yes	yes	no
pwr5	yes	yes	yes	no
pwr5x	yes	yes	yes	no
ppc	no	no	no	no
ppc64	no	no	yes	no
ppc64gr	yes	no	yes	no
ppc64grsq	yes	yes	yes	no
ppc64v	yes	yes	yes	VMX
ppc970	yes	yes	yes	VMX
pwr6	yes	yes	yes	VMX
pwr6e	yes	yes	yes	VMX
pwr7	yes	yes	yes	VMX, VSX

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

For any given **-qarch** setting, the compiler defaults to a specific, matching **-qtune** setting, which can provide additional performance improvements. Alternatively, if you specify **-qarch** with a group argument, you can specify **-qtune** as either **auto** or provide a specific architecture in the group. For detailed information on using **-qarch** and **-qtune** together, see “-qtune” on page 260.

Specifying **-q64** changes the effective **-qarch** setting as follows:

Original -qarch setting	Effective setting when -q64 is specified
ppc	ppc64grsq
ppcgr	ppc64grsq

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

Examples

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

```
xlf -o testing myprogram.f -q32 -qarch=ppc
```

Related information

- “Compiling for specific architectures” on page 31
- **-qfloat**
- **-qprefetch**
- “-qtune” on page 260
- “Choosing the best **-qarch** suboption” in the *XL Fortran Optimization and Programming Guide*

-qassert

Category

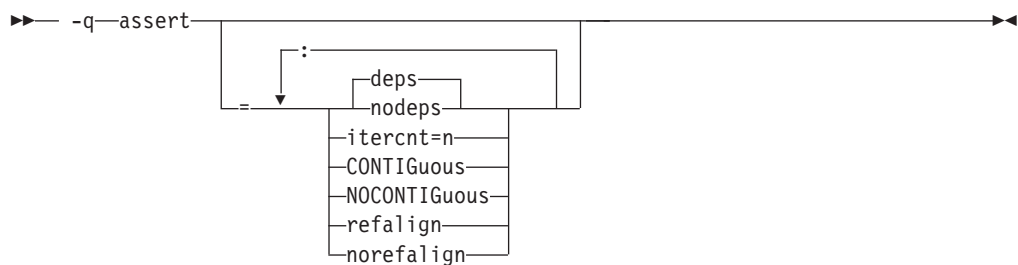
Optimization and tuning

Purpose

Provides information about the characteristics of your code that can help the compiler fine-tune optimizations.

Syntax

Option:



@PROCESS:

@PROCESS ASSERT(suboptions)

Defaults

-qassert=deps:norefalign:nocontig

Parameters

deps | nodeps

Specifies whether or not any loop-carried dependencies exist.

itercnt=*n*

Specifies a value for unknown loop iteration counts for the optimizer to use when it cannot statically determine the loop iteration count.

CONTIGuous | NOCONTIGuous

Specifies for all compilation units that:

- All array pointers are pointer associated with contiguous targets.
- All assumed-shape arrays are argument associated with contiguous actual arguments.

The **contig** suboption allows the compiler to perform optimizations according to the memory layout of the objects occupying contiguous blocks of memory. Use this suboption with discretion as it may produce unexpected results without warning.

Note: The contiguous suboption is not supported through the **ASSERT** directive.

refalign | norefalign

Specifies that all pointers inside the compilation unit only point to data that is naturally aligned according to the length of the pointer types. With

this assertion, the compiler might generate more efficient code. This assertion is particularly useful when you target a SIMD architecture with `-qhot=level=0` or `-qhot=level=1` with `-qsimd=auto`.

Related information

- *High-order transformation* in the *XL Fortran Optimization and Programming Guide* for background information and instructions on using these assertions.
- The `ASSERT` directive in the *XL Fortran Language Reference*.

-qattr

Category

Listings, messages, and compiler information

Purpose

Produces a compiler listing that includes the attribute component of the attribute and cross-reference section of the listing.

Syntax



@PROCESS:

`@PROCESS ATTR[(FULL)] | NOATTR`

Defaults

`-qnoattr`

Parameters

full

Reports all identifiers in the program, whether they are referenced or not. If you specify `-qattr` without this suboption, reports only those identifiers that are used.

Usage

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

- “Listings, messages, and compiler information” on page 85
- “Attribute and cross reference section” on page 305

-qautodbl

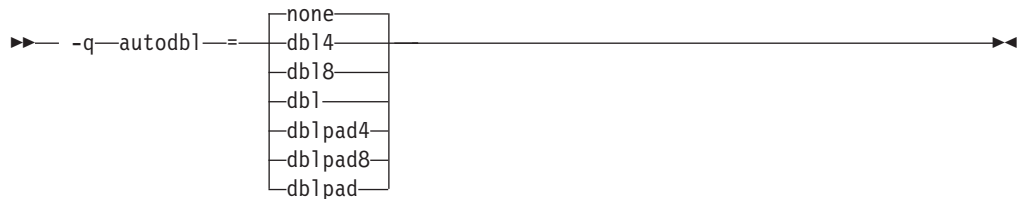
Category

Floating-point and integer control

Purpose

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

Syntax



@PROCESS:

@PROCESS AUTODBL(*setting*)

Defaults

-qautodbl=none

Parameters

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:

none Does not promote or pad any objects that share storage. This setting is the default.

db14 Promotes floating-point objects that are single-precision (4 bytes in size) or that are composed of such objects (for example, **COMPLEX** or array objects):

- **REAL(4)** is promoted to **REAL(8)**.
- **COMPLEX(4)** is promoted to **COMPLEX(8)**.

This suboption requires the **libxlfpm4.a** library during linking.

db18 Promotes floating-point objects that are double-precision (8 bytes in size) or that are composed of such objects:

- **REAL(8)** is promoted to **REAL(16)**.
- **COMPLEX(8)** is promoted to **COMPLEX(16)**.

This suboption requires the **libxlfpm8.a** library during linking.

dbl Combines the promotions that **db14** and **db18** perform.

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

dblpad4

Performs the same promotions as **dbl4** and pads objects of other types (except **CHARACTER**) if they could possibly share storage with promoted objects.

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

dblpad8

Performs the same promotions as **dbl8** and pads objects of other types (except **CHARACTER**) if they could possibly share storage with promoted objects.

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

dblpad

Combines the promotions done by **dbl4** and **dbl8** and pads objects of other types (except **CHARACTER**) if they could possibly share storage with promoted objects.

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

Usage

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.

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**.

You must not specify the **-qautodbl** option if your program contains vector types.

Restrictions

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

Related information

For background information on promotion, padding, and storage/value relationships and for some source examples, see “Implementation details for -qautodbl promotion and padding” on page 310.

“-qrealsize” on page 223 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.

-qbindcextname

Category

Portability and migration

Purpose

Controls whether the **-qextname** option affects **BIND(C)** entities.

Syntax

►► -q bindcextname
nobindcextname ►►

@PROCESS:

@PROCESS **BINDCEXTNAME** | NOBINDCEXTNAME

Defaults

-qbindcextname

Usage

The **-qextname** option and the **BIND(C)** attribute are two ways of modifying the names of Fortran global entities to facilitate use in C.

If you explicitly specify a **BIND(C)** binding label in an interface block using the **NAME=** specifier, the compiler uses this binding label in calls to the procedure regardless of the **-qextname** and **-qbindcextname** options.

If your interface does not explicitly specify a **BIND(C)** binding label using the **NAME=** specifier, the compiler creates an implicit binding label. If you also specify the **-qextname** option, the compiler appends an underscore to the implicit binding label only when the **-qbindcextname** option is in effect.

If you specify the **-qextname** and **-qbindcextname** options for a compilation unit declaring a **BIND(C)** procedure, the compiler appends an underscore to the binding label, even when the binding label is explicitly specified.

Notes:

- You must ensure that the names of a **BIND(C)** entity are identical. Accordingly, if two compilation units access the same **BIND(C)** entity that does not have an explicitly-specified binding label, you must not compile one unit with the **-qbindcextname** option and the other with the **-qnobindcextname** option.
- The **-q[no]bindcextname** option has effect only if the **-qextname** option is also specified. If the **-qextname** option is specified with a list of named entities, the **-q[no]bindcextname** option only affects these named entities.

Examples

```
interface
  integer function foo() bind(c)
  end function
  integer function bar()
  end function
end interface
```

```
print *, foo()
print *, bar()
end
```

```
xlf90 x.f -qextname -qbindcextname      # calls "foo_", and "bar_"
xlf90 x.f -qextname -qnobindcextname    # calls "foo", and "bar"
xlf90 x.f -qextname=foo -qbindcextname  # calls "foo_", and "bar_"
xlf90 x.f -qextname=foo -qnobindcextname # calls "foo", and "bar"
xlf90 x.f                               # calls "foo", and "bar"
xlf90 x.f -qnobindcextname              # calls "foo", and "bar"
```

Related information

- “-qextname” on page 148
- "BIND (Fortran 2003)"
- "Binding labels"

-qcache

Category

Optimization and tuning

@PROCESS

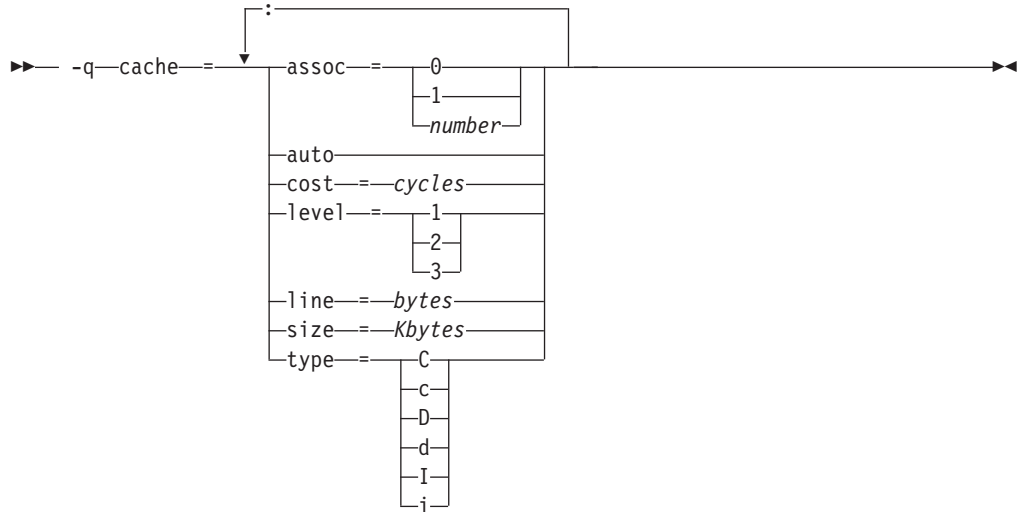
None.

Purpose

When specified with **-O4**, **-O5**, or **-qipa**, 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.

Syntax



Defaults

Not applicable.

Parameters

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

Usage

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 the **-qhot** option or an option that implies **-qhot**.

- 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.

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.

Examples

To tune performance for a system with a combined instruction and data level-1 cache where the cache is two-way associative, 8 KB in size, and has 64-byte cache lines:

```
xlf95 -O3 -qhot -qcache=type=c:level=1:size=8:line=64:assoc=2 file.f
```

To tune performance for a system with two levels of data cache, use two **-qcache** options:

```
xlf95 -O3 -qhot -qcache=type=D:level=1:size=256:line=256:assoc=4 \  
-qcache=type=D:level=2:size=512:line=256:assoc=2 file.f
```

To tune performance for a system with two types of cache, again use two **-qcache** options:

```
xlf95 -O3 -qhot -qcache=type=D:level=1:size=256:line=256:assoc=4 \  
-qcache=type=I:level=1:size=512:line=256:assoc=2 file.f
```

Related information

- “-qarch” on page 120
- “-qhot” on page 167
- “-qtune” on page 260

-qcclines

Category

Input control

Purpose

Determines whether the compiler recognizes conditional compilation lines in fixed source form and F90 free source form. This option is not supported with IBM free source form.

Syntax

►► -q cclines
nocclines ◄◄

@PROCESS:

@PROCESS CCLINES | NOCCLINES

Defaults

The default is **-qcclines** if the **-qsmp=omp** option is turned on; otherwise, the default is **-qnocclines**.

Related information

- *Conditional compilation in the XL Fortran Language Reference*

-qcheck

Category

Error checking and debugging

Purpose

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

Syntax

►► -q nocheck
check ◄◄

@PROCESS:

@PROCESS CHECK | NOCHECK

Defaults

-qnocheck

-qci

Category

Input control

Purpose

Specifies the identification numbers (from 1 to 255) of the **INCLUDE** lines to process.

Syntax

►► -qci=number ◀◀

@PROCESS:

@PROCESS CI(*number*,...,*number*)

Defaults

Not applicable.

Usage

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.

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.

Note:

1. Because the optional number in **INCLUDE** lines is not a widespread XL Fortran feature, using it may restrict the portability of a program.
2. This option works only with the XL Fortran **INCLUDE** directive and not with the **#include** C preprocessor directive.

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.

Related information

- The `INCLUDE` directive in the *XL Fortran Language Reference*

-qcompact

Category

Optimization and tuning

Purpose

Avoids optimizations that increase code size.

Syntax

►► -q nocompact
compact ◄◄

@PROCESS:

@PROCESS COMPACT | NOCOMPACT

Defaults

-qnocompact

Usage

By default, some techniques the optimizer uses to improve performance, such as loop unrolling and array vectorization, may also make the program larger. For systems with limited storage, you can use **-qcompact** to reduce the expansion that takes place. If your program has many loop and array language constructs, using the **-qcompact** option will affect your application's overall performance. You may want to restrict using this option to those parts of your program where optimization gains will remain unaffected.

Rules

With **-qcompact** in effect, other optimization options still work; the reductions in code size come from limiting code replication that is done automatically during optimization.

-qcr

Category

Input control

@PROCESS

None.

Purpose

Controls how the compiler interprets the CR (carriage return) character.

This option allows you to compile code written using a Mac OS or DOS/Windows® editor.

Syntax

►► -q no cr

Defaults

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.

Usage

If you specify **-qno cr**, the compiler recognizes only the LF character as a line terminator. You must specify **-qno cr** if you use the CR character for a purpose other than line termination.

-qctyp lss

Category

Portability and migration

Purpose

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.

Syntax

►► -q noctyp lss
ctyp lss no arg
arg

@PROCESS:

@PROCESS CTYPLSS[([NO]ARG)] | NOCTYPLSS

Defaults

-qnoctyp lss

Parameters

arg | no arg

Suboptions retain the behavior of **-qctyp lss**. Additionally, **arg** specifies that Hollerith constants used as actual arguments will be treated as integer actual arguments.

Usage

With `-qctyplss`, character constant expressions are treated as if they were Hollerith constants and thus can be used in logical and arithmetic expressions.

- If you specify the `-qctyplss` 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 `-qctyplss` 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

- *Hollerith constants in the XL Fortran Language Reference*
- *Passing arguments by reference or by value in the XL Fortran Optimization and Programming Guide*

-qdbg

Category

Error checking and debugging

Purpose

`-qdbg` is the long form of “-g” on page 103.

Syntax

►► — -q — nodbg
dbg —————►►

@PROCESS:

@PROCESS DBG | NODBG

Defaults

`-qnodbg`

-qddim

Category

Portability and migration

Purpose

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.

Syntax

►► — -q — noddim
ddim —————►►

@PROCESS:

@PROCESS DDIM | NODDIM

Defaults

`-qnoddim`

Usage

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
```

-qdescriptor

Category

Portability and migration

@PROCESS

None.

Purpose

Specifies the XL Fortran internal descriptor data structure format to use for non object-oriented entities in your compiled applications.

Syntax

→ -q-descriptor=v1/v2 →

Defaults

- -qdescriptor=v1

Parameters

- v1** Use an internal descriptor data structure format that provides compatibility with objects compiled with XL Fortran V10.1 and earlier.
- v2** Use an internal descriptor data structure format that provides compatibility with new features available in XL Fortran V11.1 and above. This setting allows your programs to take advantage of new object-oriented features and constructs.

Usage

Regardless of what **-qdescriptor** setting is in effect, applications containing object-oriented constructs will use the **v2** data structure format for those constructs, and will not be compatible with objects compiled with XL Fortran V10.1 or earlier.

You should consider explicitly using the **v2** setting if your applications do not need to interact with objects that were compiled with earlier versions of XL Fortran.

The choice of **-qdescriptor** setting is an important consideration when building libraries or modules for distribution. Users of these libraries and modules will need to be aware of the **-qdescriptor** setting and compile the code that uses them in a compatible way. It is suggested that such libraries and modules be built with the **-qsaveopt** option so that the objects themselves will encode the compilation options in a user-readable form.

If you are building modules with V11.1 or later that contain user-visible derived types, consider building them with the **-qxf2003=polymorphic** suboption. This allows users of the module to use or extend the derived types in a Fortran object-oriented context that uses polymorphism.

In the Fortran 2003 object-oriented programming model, the XL Fortran compiler supports using types and type extensions from types defined in modules not compiled with **-qxf2003=polymorphic**, as long as the types are not used in a context that requires polymorphism. This support extends to modules built with older XL Fortran compilers, as well. However, if the compiler detects the attempted use of a type or a type extension from a module not compiled with **-qxf2003=polymorphic** in a context that requires polymorphism, an error message will be issued and compilation halted.

If a module built with the **-qdescriptor=v1** setting or a module built with XL Fortran v10.1 or earlier is used in a compilation where **-qdescriptor=v2** has been specified, the compiler will diagnose this mismatch and halt compilation after issuing an error message.

When using the **-qdescriptor=v2** option, the compiler is unable to diagnose unsafe usage where objects built with the **v2** setting are mixed with those built with the **v1** setting or with XL Fortran 10.1 or older compilers. Even if your program appears to function properly, this usage is unsupported. The descriptor formats are different sizes and, when used with certain constructs, data layouts will change resulting in undefined and unsupported behavior. For example, the sizes of allocatable and pointer entities within derived types will be different resulting a differing size for the derived type itself.

Related information

- “-qsaveopt” on page 230
- “-qxf2003” on page 272

-qdirective

Category

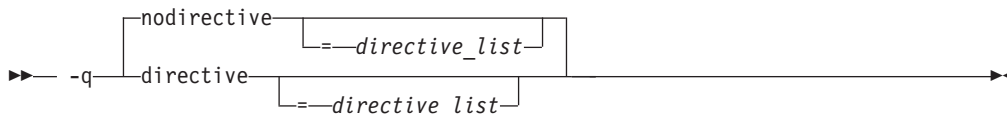
Input control

Purpose

Specifies sequences of characters, known as trigger constants, that identify comment lines as compiler comment directives.

A compiler comment directive is a line that is not a Fortran statement but is recognized and acted on by the compiler.

Format



@PROCESS:

@PROCESS DIRECTIVE[(*directive_list*)] | **NODIRECTIVE**[(*directive_list*)]

Defaults

The compiler recognizes the default trigger constant **IBM***.

Specifying **-qsmp** implies **-qdirective=smp\\$\:\$omp:ibmp**, and, by default, the trigger constants **SMP\$**, **\$OMP**, and **IBMP** are also turned on. If you specify **-qsmp=omp**, the compiler ignores all trigger constants that you have specified up to that point and recognizes only the **\$OMP** trigger constant. Specifying **-qthreaded** implies **-qdirective=ibmt**, and, by default, the trigger constant **IBMT** is also turned on.

Parameters

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**.

Usage

Note the following:

- 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.
- As the use of incorrect trigger constants can generate warning messages, error messages, or both, you should check the particular directive statement for the suitable associated trigger constant.

Examples

```
! This program is written in Fortran free source form.
PROGRAM DIRECTV
INTEGER A, B, C, D, E, F
A = 1 ! Begin in free source form.
B = 2
```

```

!OLDSTYLE SOURCEFORM(FIXED)
! Switch to fixed source form for this include file.
    INCLUDE 'set_c_and_d.inc'
!IBM* SOURCEFORM(FREE)
! Switch back to free source form.
E = 5
F = 6
END

```

For this example, compile with the option **-qdirective=oldstyle** to ensure that the compiler recognizes the **SOURCEFORM** directive before the **INCLUDE** line. After processing the include-file line, the program reverts back to free source form, after the **SOURCEFORM(FREE)** statement.

- The **SOURCEFORM** directive in the *XL Fortran Language Reference*
- The *Directives* section in the *XL Fortran Language Reference*

-qdirectstorage

Category

Optimization and tuning

@PROCESS

None.

Context

None.

Purpose

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

Format

Defaults

-qnodirectstorage

Usage

Use this option with discretion. It is intended for programmers who know how the memory and cache blocks work, and how to tune their applications for optimal performance. For a program to execute correctly on all Power implementations of cache organization, the programmer should assume that separate instruction and data caches exist, and should program to the separate cache model.

Note: Using the **-qdirectstorage** option together with the **CACHE_ZERO** directive may cause your program to fail, or to produce incorrect results.

Related information

- `CACHE_ZERO` in the *XL Fortran Language Reference*.

-qdlines

Category

Input control

Purpose

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

Format

►► -q nodlines
dlines ◀◀

@PROCESS:

@PROCESS D LINES | NODLINES

Defaults

`-qnodlines`

-qdpc

Category

Floating-point and integer control

Purpose

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.

Format

►► -q nodpc
dpc _=_e ◀◀

@PROCESS:

@PROCESS DPC[(E)] | NODPC

Defaults

`-qnodpc`

Usage

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.

`-qautodbl` and `-qrealsize` are more general-purpose options that can also do what `-qdp` does. `-qdp` has no effect if you specify either of these options.

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    1.0000000000000000    T
1.000000000    1.0000000000100009    F
```

showing that with `-qdp` the extra precision is preserved.

- “`-qautodbl`” on page 126
- “`-qrealsize`” on page 223

-qenum

Category

Floating-point and integer control

@PROCESS

None.

Purpose

Specifies the range of the enumerator constant and enables storage size to be determined.

Syntax



Defaults

`-qenum=4`

Usage

Regardless of its storage size, the enumerator's value will be limited by the range that corresponds to *value*. If the enumerator value exceeds the range specified, a warning message is issued and truncation is performed as necessary.

The range limit and kind type parameter corresponding to each *value* is as follows:

Table 19. Enumerator sizes and types

Value	Valid range of enumerator constant value	Kind type parameter
1	-128 to 127	4
2	-32768 to 32767	4
4	-2147483648 to 2147483647	4
8	-9223372036854775808 to 9223372036854775807	8

Related information

- *ENUM/ENDENUM* statement in the *XL Fortran Language Reference*

-qescape

Category

Portability and migration

Purpose

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.

Syntax



@PROCESS:

@PROCESS **ESCAPE** | NOESCAPE

Defaults

-qescape

Usage

When **-qescape** is specified, 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\fg'
      END
$ xlf95 escape.f
** _main === End of Compilation 1 ===
1501-510  Compilation successful for file escape.f.
$ a.out
cde
      g
$ xlf95 -qnoescape escape.f
** _main === End of Compilation 1 ===
1501-510  Compilation successful for file escape.f.
$ a.out
a\bcd\fg
```

In the first compilation, with the default setting of **-qescape**, `\b` is printed as a backspace, and `\f` is printed as a formfeed character.

With the **-qnoescape** option specified, the backslashes are printed like any other character.

Related information

The list of escape sequences that XL Fortran recognizes is shown in *Escape sequences for character strings* in the *XL Fortran Optimization and Programming Guide*.

-qessl

Category

Optimization and tuning

@PROCESS

None.

Purpose

Allows the compiler to substitute the Engineering and Scientific Subroutine Library (ESSL) routines in place of Fortran 90 intrinsic procedures.

The ESSL is a collection of subroutines that provides a wide range of mathematical functions for various scientific and engineering applications. The subroutines are tuned for performance on specific architectures. Some of the Fortran 90 intrinsic procedures have similar counterparts in ESSL. Performance is improved when these Fortran 90 intrinsic procedures are linked with ESSL. In this case, you can keep the interface of Fortran 90 intrinsic procedures, and get the added benefit of improved performance using ESSL.

Syntax

►► — -q —

noessl
essl

 —————►►

Defaults

-qnoessl

Usage

Use the ESSL Serial Library when linking with **-lessl**. Use the ESSL SMP Library when linking with **-lesslsmpl**.

-lessl or **-lesslsmpl** must be used whenever code is being compiled with **-qessl**. ESSL V4.1.1 or above is recommended. It supports both 32-bit and 64-bit environments.

Also, since **libessl.so** and **libesslsmpl.so** have a dependency on **libxlf90_r.so**, compile with **xlf_r**, **xlf90_r**, or **xlf95_r**, which use **libxlf90_r.so** as the default to link. You can also specify **-lxlf90_r** on the link command line if you use the linker directly, or other commands to link.

The following MATMUL function calls may use ESSL routines when **-qessl** is enabled:

```
real a(10,10), b(10,10), c(10,10)
      c=MATMUL(a,b)
```

Related information

The ESSL libraries are not shipped with the XL Fortran compiler. For more information about these libraries, see the Engineering and Scientific Subroutine

-qextern

Category

Portability and migration

@PROCESS

None.

Purpose

Allows user-written procedures to be called instead of XL Fortran intrinsics.

Syntax

▶▶ `-qextern=names` ▶▶

Defaults

Not applicable.

Parameters

names

A list of procedure names separated by colons.

Usage

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.

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.

-qextern

Category

Portability and migration

Purpose

Adds an underscore to the names of all global entities.

Syntax



@PROCESS:

@PROCESS EXTNAME[(*name1*, *name2*,...)] | NOEXTNAME

Defaults

-qnoextern

Parameters

name

Identifies a specific global entity (or entities). For a list of named entities, separate each name with a colon. For example: *name1: name2:...*

The name of a main program is not affected.

Usage

The **-qextern** 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

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.

This option also affects the names that are specified in the `-qextern`, `-qinline`, and `-qsigtrap` options. You do not have to include underscores in their names on the command line.

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

- “`-qextern`” on page 147
- “`-qinline`” on page 176
- “`-qsigtrap`” on page 234
- “`-qbindcextname`” on page 128

-qfdpr

Category

Optimization and tuning

@PROCESS

None.

Purpose

Provides object files with information that the IBM Feedback Directed Program Restructuring (FDPR) performance-tuning utility needs to optimize the resulting executable file.

When **-qfdpr** is in effect, optimization data is stored in the object file.

Syntax

►► -q nofdpr
fdpr ◀◀

Defaults

-qnofdpr

Usage

For best results, use **-qfdpr** for all object files in a program; FDPR will perform optimizations only on the files compiled with **-qfdpr**, and not library code, even if it is statically linked.

The optimizations that the FDPR utility performs are similar to those that the **-qpdf** option performs.

The FDPR performance-tuning utility has its own set of restrictions, and it is not guaranteed to speed up all programs or produce executables that produce exactly the same results as the original programs.

Examples

To compile `myprogram.f` so it includes data required by the FDPR utility, enter:

```
xlf myprogram.f -qfdpr
```

Related information

- “-qpdf1, -qpdf2” on page 208

-qfixed

Category

Input control

Purpose

Indicates that the input source program is in fixed source form and optionally specifies the maximum line length.

Syntax

►► -q-fixed _right_margin ◀◀

@PROCESS:

@PROCESS FIXED[(*right_margin*)]

Defaults

-qfixed=72 is the default for the xlf, xlf_r, f77, and fort77

-qfree=f90 is the default for the f90, f95, xlf90, xlf90_r, xlf95, xlf95_r, f2003, xlf2003, and xlf2003_r commands.

Usage

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.

Related information

- “-qfree” on page 159
- See *Fixed source form* in the *XL Fortran Language Reference*.

-qflag

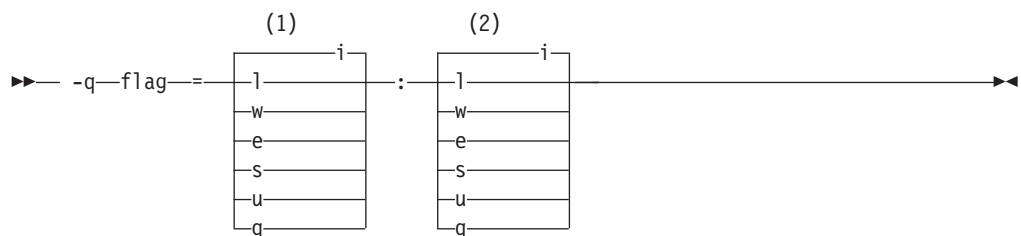
Category

Listings, messages, and compiler information

Purpose

Limits the diagnostic messages to those of a specified severity level or higher.

Syntax



Notes:

- 1 Minimum severity level of messages reported in listing
- 2 Minimum severity level of messages reported on terminal

@PROCESS:

@PROCESS FLAG(*listing_severity,terminal_severity*)

Defaults

`-qflag=i:i`, which shows all compiler messages.

Parameters

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.

Usage

You must specify both *listing_severity* and *terminal_severity*.

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.

The `-qflag` option overrides any `-qlanglvl` or `-qsaa` options specified.

The `-w` option is a short form for `-qflag=e:e`.

Related information

- “`-qhalt`” on page 166
- “`-qlanglvl`” on page 187
- “`-qsaa`” on page 228
- “`-qsuppress`” on page 254
- “`-w`” on page 286
- “Understanding XL Fortran error messages” on page 291

-qfloat

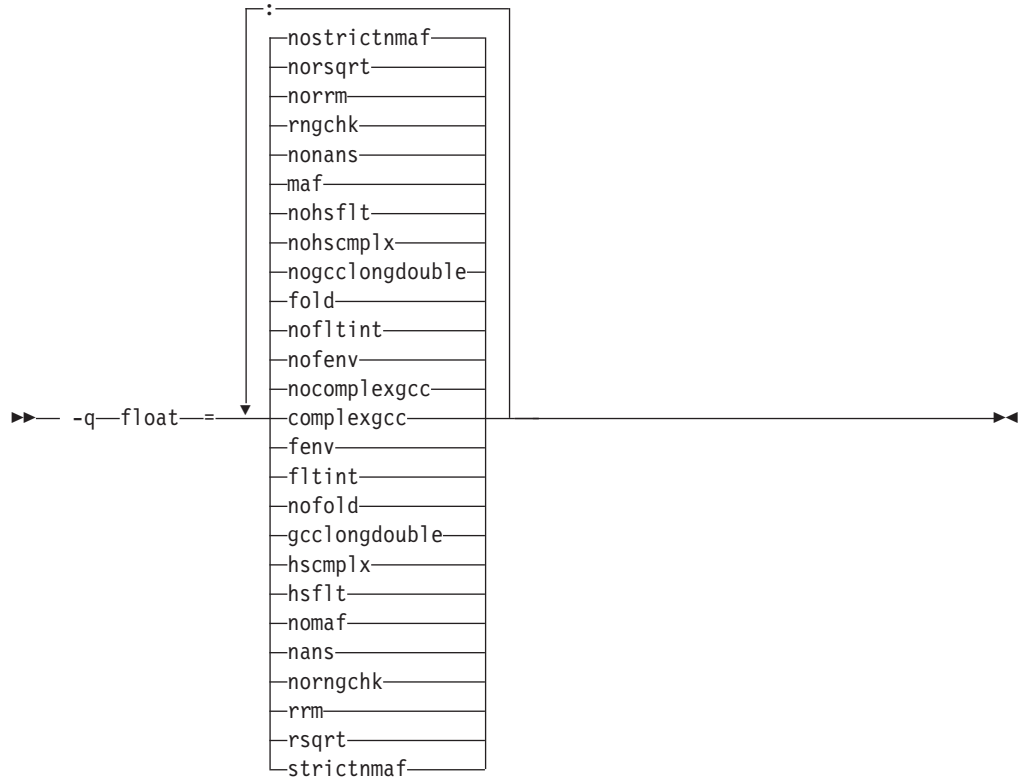
Category

Floating-point and integer control

Purpose

Selects different strategies for speeding up or improving the accuracy of floating-point calculations.

Syntax



@PROCESS:

@PROCESS FLOAT(*suboptions*)

Defaults

- `-qfloat=nocomplexgcc:nofenv:nofltint:fold:nogcclongdouble:nohscmplx:nohsflt:maf:nonans:rngchk:norrm:norsqrt:nostrictnmaf`
- `-qfloat=fltint:rsqrt:norngchk` when `-qnostrict`, `-qstrict=nooperationprecision:noexceptions`, or `-O3` or higher optimization level is in effect.
- `-qfloat=nocomplexgcc` when 64-bit mode is enabled.

Parameters

`complexgcc` | `nocomplexgcc`

Specifies whether GCC conventions for passing or returning complex numbers are to be used. `complexgcc` preserves compatibility with GCC-compiled code and the default setting of IBM XL C/C++ compilers.

Note: For this suboption, restrict intermixing of XL Fortran-compiled code with non-XL Fortran-compiled code to small, self-contained, mathematically-oriented subprograms that do not rely on any runtime-library

information or global data, such as module variables. Do not expect exception handling or I/O to work smoothly across programs compiled from different environments.

fenv | **nofenv**

Specifies whether the code depends on the hardware environment and whether to suppress optimizations that could cause unexpected results due to this dependency.

Certain floating-point operations rely on the status of Floating-Point Status and Control Register (FPSCR), for example, to control the rounding mode or to detect underflow. In particular, many compiler built-in functions read values directly from the FPSCR.

When **nofenv** is in effect, the compiler assumes that the program does not depend on the hardware environment, and that aggressive compiler optimizations that change the sequence of floating-point operations are allowed. When **fenv** is in effect, such optimizations are suppressed.

You should use **fenv** for any code containing statements that read or set the hardware floating-point environment, to guard against optimizations that could cause unexpected behavior.

fltint | **nofltint**

Speeds up floating-point-to-integer conversions by using an inline sequence of code instead of a call to a library function. The library function, which is called when **nofltint** is in effect, checks for floating-point values outside the representable range of integers and returns the minimum or maximum representable integer if passed an out-of-range floating-point value.

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.

If **-qarch** is set to a processor that has an instruction to convert from floating point to integer, that instruction will be used regardless of the **[no]fltint** setting. This conversion also applies to all Power processors in 64-bit mode.

If you compile with **-O3** or higher optimization level, **fltint** is enabled automatically. To disable it, also specify **-qstrict**, **-qstrict=operationprecision**, or **-qstrict=exceptions**.

fold | **nofold**

Evaluates constant floating-point expressions at compile time, which may yield slightly different results from evaluating them at run time. The compiler always evaluates constant expressions in specification statements, even if you specify **nofold**.

gcclongdouble | **nogcclongdouble**

Specifies whether the compiler uses GCC-supplied or IBM-supplied library functions for 128-bit REAL(16) operations.

gcclongdouble ensures binary compatibility with GCC for mathematical calculations. If this compatibility is not important in your application, you should use **nogcclongdouble** for better performance. This suboption only has an effect when 128-bit long double types are enabled with **-qldb128**.

Note: Passing results from modules compiled with **nogcclongdouble** to modules compiled with **gcclongdouble** may produce different results for numbers such as Inf, NaN and other rare cases. To avoid such

incompatibilities, the compiler provides built-in functions to convert IBM long double types to GCC long double types; see for more information.

hscmplx | **nohscmplx**

Speeds up operations involving complex division and complex absolute value. This suboption, which provides a subset of the optimizations of the **hsflt** suboption, is preferred for complex calculations.

hsflt | **nohsflt**

Speeds up calculations by preventing rounding for single-precision expressions and by replacing floating-point division by multiplication with the reciprocal of the divisor. It also uses the same technique as the **fltint** suboption for floating-point-to-integer conversions. **hsflt** implies **hscmplx**.

The **hsflt** suboption overrides the **nans** and **spnans** suboptions.

Note: Use **-qfloat=hsflt** on applications that perform complex division and floating-point conversions where floating-point calculations have known characteristics. In particular, all floating-point results must be within the defined range of representation of single precision. Use with discretion, as this option may produce unexpected results without warning. For complex computations, it is recommended that you use the **hscmplx** suboption (described above), which provides equivalent speed-up without the undesirable results of **hsflt**.

maf | **nomaf**

Makes floating-point calculations faster and more accurate by using floating-point multiply-add instructions where appropriate. The results may not be exactly equivalent to those from similar calculations performed at compile time or on other types of computers. Negative zero results may be produced. This suboption may affect the precision of floating-point intermediate results. If **-qfloat=nomaf** is specified, no multiply-add instructions will be generated unless they are required for correctness.

nans | **nonans**

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

rngchk | **norngchk**

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

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

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

If any of these conditions are not met, incorrect results may be produced. For example, if the divisor for a division operation is 0.0 or a denormalized number (absolute value $< 2^{-1022}$ for double precision, and absolute value $< 2^{-126}$

for single precision), NaN, instead of INF, may result; when the divisor is +/- INF, NaN instead of 0.0 may result. If the input is +INF for a sqrt operation, NaN, rather than INF, may result.

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

rrm | **norrm**

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

rsqrt | **norsqrt**

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

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

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**.

Note:

- For details about the relationship between **-qfloat** suboptions and their **-qstrict** counterparts, see “-qstrict” on page 247.

Usage

Using **-qfloat** suboptions other than the default settings may produce incorrect results in floating-point computations if not all required conditions for a given suboption are met. For this reason, you should only use this option if you are experienced with floating-point calculations involving IEEE floating-point values and can properly assess the possibility of introducing errors in your program. See also “Implementation details of XL Fortran floating-point processing” in the *XL Fortran Optimization and Programming Guide* for more information.

If the **-qstrict** | **-qnostrict** and **float** suboptions conflict, the last setting specified is used.

Examples

To compile `myprogram.f` so that constant floating point expressions are evaluated at compile time and multiply-add instructions are not generated, enter:

```
xlf myprogram.f -qfloat=fold:nomaf
```

Related information

- “-qarch” on page 120
- “-qfltrap” on page 158
- “-qstrict” on page 247

-qfpp

Category

Input control

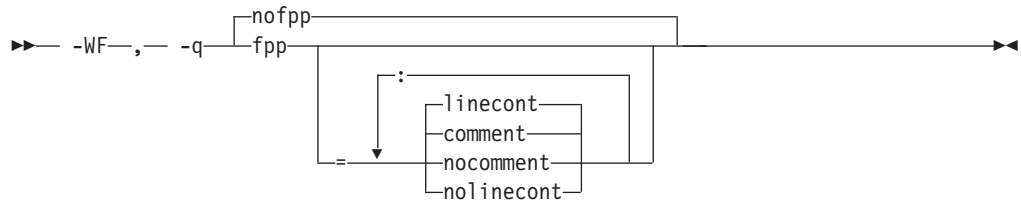
@PROCESS

None.

Purpose

Controls Fortran-specific preprocessing in the C preprocessor.

Syntax



Defaults

- -qnofpp

Parameters

comment | nocomment

Instructs the C preprocessor (**cpp**) to recognize the ! character as a comment delimiter in macro expansion. When this suboption is enabled, the ! character and all characters following it on that line will be ignored by **cpp** when performing macro expansion.

linecont | nolinecont

Instructs **cpp** to recognize the & character as a line continuation character. When this suboption is enabled, **cpp** treats the & character and the C-style \ line continuation character equivalently.

Specifying **-qfpp** without any suboptions is equivalent to **-qfpp=comment:linecont**.

Usage

-qfpp is a C preprocessor option, and must therefore be specified using the **-WF** option.

Related information

- “-W” on page 284
- “-qppsuborigarg” on page 219
- “Passing Fortran files through the C preprocessor” on page 31

-qflttrap

Category

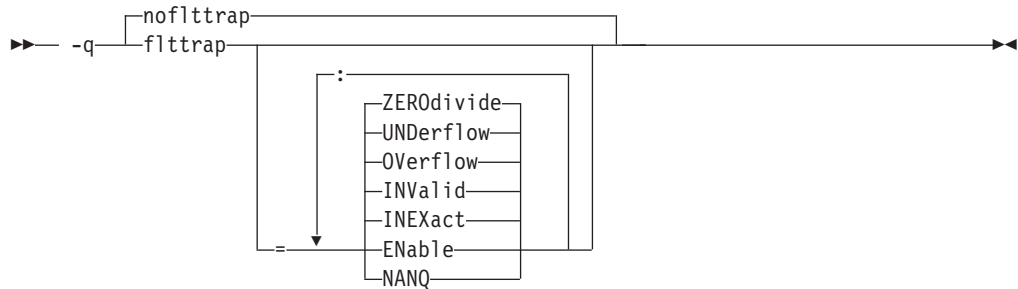
Error checking and debugging

Purpose

Determines what types of floating-point exception conditions to detect at run time.

The program receives a **SIGFPE** signal when the corresponding exception occurs.

Syntax



@PROCESS:

FLTTRAP[(*suboptions*)] | NOFLTTRAP

Defaults

-qnoflttrap

Parameters

ENable

Turn on checking for the specified exceptions in the main program so that the exceptions generate **SIGFPE** signals. You must specify this suboption if you want to turn on exception trapping without modifying your source code.

IMPrecise

Only check for the specified exceptions on subprogram entry and exit. This suboption improves performance, but it can make the exact spot of the exception difficult to find.

INEXact

Detect and trap on floating-point inexact if exception-checking is enabled. Because inexact results are very common in floating-point calculations, you usually should not need to turn this type of exception on.

INValid

Detect and trap on floating-point invalid operations if exception-checking is enabled.

NANQ

Detect and trap all quiet not-a-number values (NaNQs) and signaling not-a-number values (NaNSs). Trapping code is generated regardless of specifying the **enable** or **imprecise** suboption. This suboption detects all

NaN values handled by or generated by floating point instructions, including those not created by invalid operations. This option can impact performance.

Overflow

Detect and trap on floating-point overflow if exception-checking is enabled.

Underflow

Detect and trap on floating-point underflow if exception-checking is enabled.

ZeroDivide

Detect and trap on floating-point division by zero if exception-checking is enabled.

Usage

Specifying **-qfltrap** without suboptions is equivalent to **-qfltrap=inv:inex:ov:und:zero**. 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.

The **-qfltrap** option is recognized during linking with IPA. Specifying the option at the link step overrides the compile-time setting.

Examples

When you compile this program:

```
REAL X, Y, Z
DATA X /5.0/, Y /0.0/
Z = X / Y
PRINT *, Z
END
```

with the command:

```
xlf95 -qfltrap=zerodivide:enable -qsigtrap divide_by_zero.f
```

the program stops when the division is performed.

The **zerodivide** suboption identifies the type of exception to guard against. The **enable** suboption causes a **SIGFPE** signal when the exception occurs. The **-qsigtrap** option produces informative output when the signal stops the program.

Related information

- “-qsigtrap” on page 234
- “-qarch” on page 120
- *Detecting and trapping floating-point exceptions in the XL Fortran Optimization and Programming Guide* has full instructions on how and when to use the **-qfltrap** option, especially if you are just starting to use it.

-qfree

Category

Input control

Purpose

Indicates that the source code is in free source form.

Syntax



@PROCESS:

```
@PROCESS FREE[({F90|IBM})]
```

Defaults

-qfree by itself specifies Fortran 90 free source form.

-qfixed=72 is the default for the **xl**, **xl_r**, **f77**, and **fort77** commands.

-qfree=f90 is the default for the **f90**, **f95**, **xl**, **xl_r**, **xl95**, **xl95_r**, **f2003**, **xl**, **xl_r**, **xl2003**, and **xl2003_r** commands.

Parameters

ibm

Specifies compatibility with the free source form defined for VS FORTRAN.

f90

Specifies compatibility with the free source form defined for Fortran 90.

Note that the free source form defined for Fortran 90 also applies to Fortran 95, and Fortran 2003.

Usage

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).

Fortran 90 free source form 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.

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

Related information

- “-qfixed” on page 150
- “-k” on page 105
- Free source form in the *XL Fortran Language Reference*

-qfullpath

Category

Error checking and debugging

@PROCESS

None.

Purpose

When used with the **-g** or **-qlinedebug** option, this option records the full, or absolute, path names of source and include files in object files compiled with debugging information, so that debugging tools can correctly locate the source files.

Syntax

►► — -q nofullpath
fullpath —►►

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.

Usage

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** or **-qlinedebug** option so that source-level debuggers can locate the correct source files.

Although **-qfullpath** works without the **-g** or **-qlinedebug** option, you cannot do source-level debugging unless you also specify the **-g** or **-qlinedebug** 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

- “-g” on page 103
- “-qlinedebug” on page 191

-qfunctrace

Category

Error checking and debugging

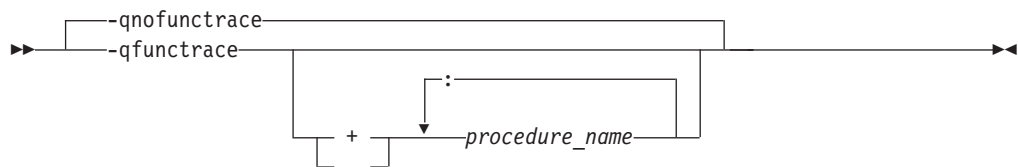
@PROCESS

None.

Purpose

Traces entry and exit points of procedures in your program. If your program contains C++ compilation units, this option also traces C++ catch blocks.

Syntax



Defaults

-qnofunctrace

Parameters

- + Instructs the compiler to trace *procedure_name* and all its internal procedures.
- Instructs the compiler not to trace *procedure_name* or any of its internal procedures.

procedure_name

The name of a program, external procedure, or module procedure. The name is case-sensitive when **-qmixed** is in effect. **BIND(C)** binding labels and mangled module procedure names are allowed, but they must have the correct case. If **-qextname** is in effect, *procedure_name* is the name of the procedure without the additional underscore.

Usage

-qfunctrace enables tracing for all procedures in your program. **-qnofunctrace** disables tracing that was enabled by **-qfunctrace**.

The **-qfunctrace+** and **-qfunctrace-** suboptions enable tracing for a specific list of procedures and are not affected by **-qnofunctrace**. The list of procedures is cumulative.

This option inserts calls to user-defined tracing procedures. These procedures must be provided at the link step. For details about the interface of tracing procedures, as well as when they are called, see the Trace procedures in your code section in the *XL Fortran Optimization and Programming Guide*.

Examples

To trace all procedures, use **-qfunctrace**.

To trace only procedures x, y, and z, use **-qfunctrace+x:y:z**.

To trace all procedures, except for x, use **-qfunctrace -qfunctrace-x**.

-qfunctrace+x -qfunctrace+y or **-qfunctrace+x -qnofunctrace -qfunctrace+y** enables tracing for only x and y.

-qfunctrace-x -qfunctrace or **-qfunctrace -qfunctrace-x** traces all procedures in the compilation unit except for x.

-qfunctrace+y -qnofunctrace traces y only.

Related information

- “-qfunctrace_xlf_catch”
- “-qfunctrace_xlf_enter” on page 164
- “-qfunctrace_xlf_exit” on page 165
- For details about the directives that you can use to specify the name of the tracing procedures, see the **FUNCTTRACE_XLF_CATCH**, **FUNCTTRACE_XLF_ENTER**, **FUNCTTRACE_XLF_EXIT** sections in the *XL Fortran Language Reference*.
- For details about the rules for using the **NOFUNCTRACE** directive, see **NOFUNCTRACE** in the *XL Fortran Language Reference*.
- For detailed information about how to implement procedure tracing routines in your code, as well as detailed examples and a list of rules for using them, see **Tracing procedures in your code** in the *XL Fortran Optimization and Programming Guide*.

-qfunctrace_xlf_catch

Category

Error checking and debugging

@PROCESS

None.

Purpose

Specifies the name of the catch tracing subroutine.

Syntax

►►—**-qfunctrace_xlf_catch**—=*catch_routine*—►►

Defaults

Not applicable.

Parameters

catch_routine

Indicates the name of the catch tracing subroutine.

Usage

You use the **-qfunctrace_xlf_catch** option to specify that the external or module procedure being compiled must be used as a catch tracing procedure.

Note:

- If you write a tracing subroutine, make sure that the program does not contain any user procedures called `__func_trace_catch`.
- You must not specify the name of an internal subroutine when you use the **-qfunctrace_xlf_catch** option.

Related information

- The **FUNCTRACE_XLF_CATCH** directive in the *XL Fortran Language Reference*.
- “-qfunctrace” on page 162
- “-qfunctrace_xlf_enter”
- “-qfunctrace_xlf_exit” on page 165
- For detailed information about how to implement the tracing procedures in your code, see **Tracing procedures in your code** in the *XL Fortran Optimization and Programming Guide*.

-qfunctrace_xlf_enter

Category

Error checking and debugging

@PROCESS

None.

Purpose

Specifies the name of the entry tracing subroutine.

Syntax

►►—qfunctrace_xlf_enter—=—enter_routine—◄◄

Defaults

Not applicable.

Parameters

enter_routine

Indicates the name of the entry tracing subroutine.

Usage

You use the **-qfunctrace_xlf_enter** option to specify that the external or module procedure being compiled must be used as an entry tracing procedure.

Note:

- If you write a tracing subroutine, make sure that the program does not contain any user procedures called `__func_trace_enter`.
- You must not specify the name of an internal subroutine when you use the `-qfunctrace_xlf_enter` option.

Related information

- The `FUNCTRACE_XLF_ENTER` directive in the *XL Fortran Language Reference*.
- “`-qfunctrace`” on page 162
- “`-qfunctrace_xlf_catch`” on page 163
- “`-qfunctrace_xlf_exit`”
- For detailed information about how to implement the tracing procedures in your code, see **Tracing procedures in your code** in the *XL Fortran Optimization and Programming Guide*.

-qfunctrace_xlf_exit**Category**

Error checking and debugging

@PROCESS

None.

Purpose

Specifies the name of the exit tracing subroutine.

Syntax

► `-qfunctrace_xlf_exit=exit_routine` ◄

Defaults

Not applicable.

Parameters

exit_routine

Indicates the name of the exit tracing subroutine.

Usage

You use the `-qfunctrace_xlf_exit` option to specify that the external or module procedure being compiled must be used as an exit tracing procedure.

Note:

- If you write a tracing subroutine, make sure that the program does not contain any user procedures called `__func_trace_exit`.
- You must not specify the name of an internal subroutine when you use the `-qfunctrace_xlf_exit` option.

Related information

- The `FUNCTRACE_XLF_EXIT` directive in the *XL Fortran Language Reference*.
- “-qfunctrace” on page 162
- “-qfunctrace_xlf_catch” on page 163
- “-qfunctrace_xlf_exit” on page 165
- For detailed information about how to implement the tracing procedures in your code, see **Tracing procedures in your code** in the *XL Fortran Optimization and Programming Guide*.

-qhalt

Category

Error checking and debugging

Purpose

Stops compilation before producing any object, executable, or assembler source files if the maximum severity of compile-time messages equals or exceeds the severity you specify.

Syntax



Notes:

- 1 Minimum severity level of messages that will prevent an object file from being created

@PROCESS:

@PROCESS HALT(*severity*)

Defaults

`-qhalt=s`, which prevents the compiler from generating an object file when compilation fails.

Parameters

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.

Usage

The **-qhalt** option can override the **-qobject** option, and **-qnoobject** can override **-qhalt**.

Related information

- “-qflag” on page 151
- “-qobject” on page 205

-qhot

Category

Optimization and tuning

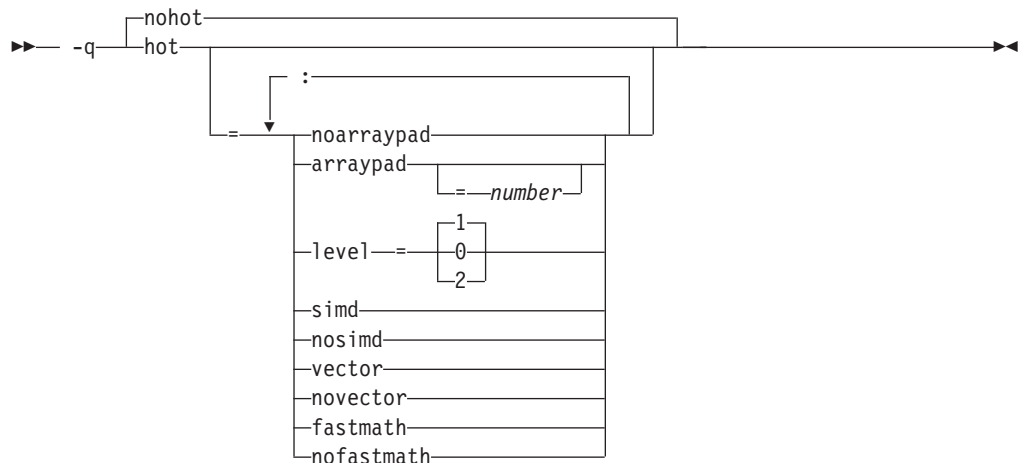
Purpose

Performs high-order loop analysis and transformations (HOT) during optimization.

The **-qhot** compiler option is a powerful alternative to hand tuning that provides opportunities to optimize loops and array language. This compiler option will always attempt to optimize loops, regardless of the suboptions you specify.

Syntax

Option :



@PROCESS:

@PROCESS HOT[=suboptions] | NOHOT

Defaults

- **-qnohot**
- **-qhot=noarraypad:level=0:novector:fastmath** when **-O3** is in effect.
- **-qhot=noarraypad:level=1:vector:fastmath** when **-qsmp**, **-O4** or **-O5** is in effect.
- Specifying **-qhot** without suboptions is equivalent to **-qhot=noarraypad:level=1:vector:fastmath**.

Parameters

arraypad | noarraypad

Permits the compiler to increase the dimensions of arrays where doing so might improve the efficiency of array-processing loops. (Because of the implementation of the cache architecture, array dimensions that are powers of two can lead to decreased cache utilization.) Specifying **-qhot=arraypad** when your source includes large arrays with dimensions that are powers of 2 can reduce cache misses and page faults that slow your array processing programs. This can be particularly effective when the first dimension is a power of 2. If you use this suboption with no *number*, the compiler will pad any arrays where it infers there may be a benefit and will pad by whatever amount it chooses. Not all arrays will necessarily be padded, and different arrays may be padded by different amounts. If you specify a *number*, the compiler will pad every array in the code.

Note: Using **arraypad** can be unsafe, as it does not perform any checking for reshaping or equivalences that may cause the code to break if padding takes place.

number

A positive integer value representing the number of elements by which each array will be padded in the source. The pad amount must be a positive integer value. It is recommended that pad values be multiples of the largest array element size, typically 4, 8, or 16.

level=0

Performs a subset of the high-order transformations and sets the default to **novector:noarraypad:fastmath**.

level=1

Performs the default set of high-order transformations.

level=2

Performs the default set of high-order transformations and some more aggressive loop transformations. **-qhot=level=2** must be used with **-qsmp**. This option performs aggressive loop analysis and transformations to improve cache reuse and exploit loop parallelization opportunities.

simd | nosimd

This suboption has been deprecated. Consider using the **-qsimd** compiler option.

vector | novector

When specified with **-qnostrict**, or an optimization level of **-O3** or higher, **vector** causes the compiler to convert certain operations that are performed in a loop on successive elements of an array (for example, square root, reciprocal square root) into a call to a routine in the Mathematical Acceleration

Subsystem (MASS) library in libxlopt. The **vector** suboption supports single and double-precision floating-point mathematics, and is useful for applications with significant mathematical processing demands.

novector disables the conversion of loop array operations into calls to MASS library routines.

Since vectorization can affect the precision of your program's results, if you are using **-O4** or higher, you should specify **-qhot=novector** if the change in precision is unacceptable to you.

fastmath | nofastmath

You can use this suboption to tune your application to either use fast scalar versions of math functions or use the default versions.

-qhot=fastmath enables the replacement of math routines with available math routines from the XLOPT library only if **-qstrict=nolibrary** is enabled.

-qhot=nofastmath disables the replacement of math routines by the XLOPT library. **-qhot=fastmath** is enabled by default if **-qhot** is specified regardless of the hot level.

Usage

If you do not also specify an optimization level when specifying **-qhot** on the command line, the compiler assumes **-O2**.

If you want to override the default **level** setting of 1 when using **-qsmp**, **-O4** or **-O5**, be sure to specify **-qhot=level=0** or **-qhot=level=2** *after* the other options.

If **-O2**, **-qnohot**, or **-qnoopt** is used on the command line, specifying HOT options in an **@PROCESS** directive will have no effect on the compilation unit.

The **-C** option turns off some array optimizations.

You can use the **-qreport** option in conjunction with **-qhot** or any optimization option that implies **-qhot** to produce a pseudo-Fortran report showing how the loops were transformed. The loop transformations are included in the listing report if either the option **-qreport** or **-qlistfmt** is also specified. This LOOP TRANSFORMATION SECTION of the listing file also contains information about data prefetch insertion locations. In addition, when you use **-qprefetch=assistthread** to generate prefetching assist threads, a message Assist thread for data prefetching was generated also appears in the LOOP TRANSFORMATION SECTION of the listing file. Specifying **-qprefetch=assitthread** guides the compiler to generate aggressive data prefetching at optimization level **-O3 -qhot** or higher. For more information, see “**-qreport**” on page 226.

Related information

- “**-qarch**” on page 120
- “**-C**” on page 98
- “**-qsimd**” on page 234
- “**-qreport**” on page 226
- “**-O**” on page 108
- “**-qstrict**” on page 247
- “**-qsmp**” on page 237
- *Using the Mathematical Acceleration Subsystem (MASS) in the XL Fortran Optimization and Programming Guide*
- *Directives for loop optimization in the XL Fortran Language Reference*
- *High-order transformation in the XL Fortran Optimization and Programming Guide*

-qieee

Category

Floating-point and integer control

Purpose

Specifies the rounding mode that the compiler will use when it evaluates constant floating-point expressions at compile time.

Syntax



@PROCESS:

```
@PROCESS IEEE({Near | Minus | Plus | Zero})
```

Defaults

Near, which rounds to the nearest representable number.

Parameters

The choices are:

Near Round to nearest representable number.

Minus Round toward minus infinity.

Plus Round toward plus infinity.

Zero Round toward zero.

Usage

Use this option 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**).

Specifying the same rounding mode for compile-time and runtime operations avoids 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 from 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.

If your program contains operations involving real(16) values, the rounding mode must be set to **-qieee=near**, round-to-nearest.

Related information

- *Selecting the rounding mode* in the *XL Fortran Optimization and Programming Guide*
- “-O” on page 108
- “-qfloat” on page 152

-qinfo

Category

Error checking and debugging

@PROCESS

None.

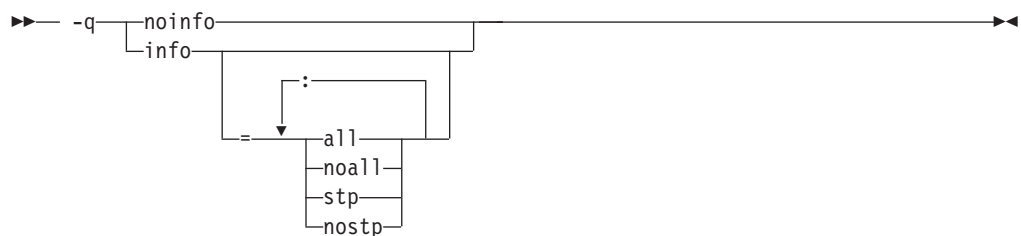
Purpose

Produces or suppresses groups of informational messages.

The messages are written to standard output and, optionally, to the listing file if one is generated.

Syntax

Option syntax



Defaults

-qnoinfo

Parameters

all Enables all diagnostic messages for all groups.

noall (option only)

Disables all diagnostic messages for all groups.

stp | nostp

Issues warnings for procedures that are not protected against stack corruption. **-qinfo=stp** has no effects unless the **-qstackprotect** option is also enabled. Like other **-qinfo** options, **-qinfo=stp** is enabled or disabled through **-qinfo=all / noall**. **-qinfo=nostp** is the default option.

Specifying **-qinfo** with no suboptions is equivalent to **-qinfo=all**.

Usage

Specifying **-qnoinfo** is equivalent to **-qinfo=noall**.

Examples

To compile `myprogram.f` to produce informational message about stack protection.

```
xlf90 myprogram.f -qinfo=stp -qstackprotect
```

Related information

- “-qreport” on page 226
- “-qstackprotect” on page 244
- For a list of deprecated options, see the “Deprecated options” on page 92 section in the *XL Fortran Compiler Reference*.

-qinit

Category

Language element control

Purpose

Makes the initial association status of pointers disassociated.

Note that this option applies to Fortran 90 and above.

Syntax

►► -q-init==f90ptr◄◄

@PROCESS:

@PROCESS INIT(F90PTR)

Defaults

Not applicable.

Usage

You can use this option to help locate and fix problems that are due to using a pointer before you define it.

Related information

- See *Pointer association* in the *XL Fortran Language Reference*.

-qinitauto

Category

Error checking and debugging

@PROCESS

None.

Purpose

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

Syntax



Defaults

-qnoinitauto

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.

Parameters

hex_value

A 1 to 8 digit hexadecimal number.

- If you do not specify a *hex_value* number, the compiler initializes the value of each byte of automatic storage to zero.
- 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.

Usage

This option 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 runtime 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 the *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 7FBFFFFF, 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.

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

- “-qflttrap” on page 158
- The **AUTOMATIC** attribute in the *XL Fortran Language Reference*

-qinlglue

Category

Object code control

Purpose

When used with **-O2** or higher optimization, inlines glue code that optimizes external function calls in your application.

Glue code, generated by the linker, is used for passing control between two external functions. When **-qinlglue** is in effect, the optimizer inlines glue code for better performance. When **-qnoinlglue** is in effect, inlining of glue code is prevented.

Note:

This option is ignored as glue code is always generated.

Syntax

►► — -q noinlglue
inlglue —►►

@PROCESS:

@PROCESS INLGLUE | NOINLGLUE

Defaults

- -qnoinlglue
- -qinlglue when -qtune=pwr4, -qtune=pwr5, -qtune=pwr6, -qtune=ppc970, -qtune=auto, or -qtune=balanced is in effect.

Usage

Inlining glue code can cause the code size to grow. Specifying **-qcompact** overrides the **-qinlglue** setting to prevent code growth. If you want **-qinlglue** to be enabled, do not specify **-qcompact**.

Specifying **-qnoinlglue** or **-qcompact** can degrade performance; use these options with discretion.

Related information

- “-q64” on page 113
- “-qcompact” on page 134
- “-qtune” on page 260
- Inlining in the *XL Fortran Optimization and Programming Guide*
- Managing code size in the *XL Fortran Optimization and Programming Guide*

-qinline

Category

Optimization and tuning

@PROCESS

None.

Purpose

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

Note:

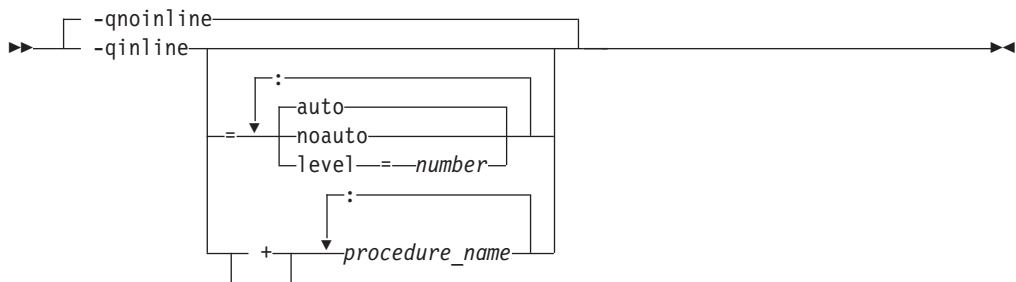
- **-qinline** replaces **-Q** and its suboptions.
- **-Q**, **-Q!**, **-Q+name**, and **-Q-name** are all deprecated options and suboptions.
- **-qipa=inline|noinline** and **-qipa=inline** suboptions **auto**, **noauto**, **limit**, and **threshold** are deprecated. **-qinline** replaces them all.

You must specify a minimum optimization level of **-O2** along with **-qinline** to enable inlining of procedures. You can also use the **-qinline** option to specify restrictions on the procedures that should or should not be inlined.

In all cases where **-qinline** is in effect, the compiler uses heuristics to determine whether inlining a specific function will result in a performance benefit. That is, whether a procedure is appropriate for inlining is subject to limits on the number of inlined calls and the amount of code size increase as a result. Therefore, simply enabling inlining does not guarantee that a given function will be inlined.

Specifying **-qnoinline** disables all inlining, including that performed by the high-level optimizer with the **-qipa** option.

Syntax



Defaults

- **-qinline=noauto:level=5**
- At an optimization level of **-O0**, **-qinline** implies **-qinline=noauto:level=5**
- At an optimization level of **-O2** or higher, **-qinline** implies **-qinline=auto:level=5**

Parameters

noauto | auto

Enables or disables automatic inlining. If you do not specify any **-qinline** suboptions, **-qinline=auto** is the default.

Note: At optimization levels of **-O2** and higher, the default is **-qinline=auto**

level=number

Provides guidance to the compiler about the relative value of inlining. The values you specify for *number* must be positive integers between 0 and 10 inclusive. The default value for *number* is 5. If you specify a value less than 5, it implies less inlining. A value greater than 5 implies more inlining than the default.

procedure_name

Indicates whether the named procedures should (after +) or should not (after -) be inlined. For example, **-qinline+foo:bar** indicates that procedures *foo* and *bar* must be inlined, and **-qinline-bar** indicates that the procedure *bar* must not be inlined. You cannot mix the "+" and "-" suboptions with each other or with other **-qinline** suboptions. For example, **-qinline+foo-bar** and **-qinline=level=5+foo** are invalid suboption combinations. However, you can use **-qinline** separately to achieve the desired effect. For example, **-qinline+foo:baz -qinline-bar -qinline=noauto:level=7**.

The **-qinline** option without any list inlines all appropriate procedures, subject to limits on the number of inlined calls and the amount of code size increase as a result. *+procedure_name* raises these limits for the named procedures.

Usage

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

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

Note: Requesting inlining for procedures with the **-qinline** option increases the possibility that those procedures are inlined. However, the compiler does not guarantee that all or a subset of those procedures are actually inlined.

Conflicting **@PROCESS** directives or compilation options applied to different compilation units can impact inlining effectiveness. For example, if you specify inlining for a procedure, some **@PROCESS** compiler directives can be rendered ineffective. See the *XL Fortran Optimization and Programming Guide* for more information about inlining and IPA.

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

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

If you specify the **-g** option to generate debug information, inlining may be suppressed.

Examples

To compile `myprogram.f` so that no functions are inlined, enter:

```
xlf myprogram.f -O2 -qnoinline
```

Assuming you have procedures `salary`, `taxes`, `expenses`, and `benefits`, to compile `myprogram.f` so that the compiler tries to inline these procedures, you enter:

```
xlf myprogram.f -O2 -qinline+salary:taxes:expenses:benefits
```

If you do not want the procedures `salary`, `taxes`, `expenses`, and `benefits` to be inlined when you compile `myprogram.f`, you enter:

```
xlf myprogram.f -O2 -qinline-salary:taxes:expenses:benefits
```

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

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

You can specify an inlining level between 6 and 10 to perform more aggressive automatic inlining. For example:

```
-O2 -qinline=auto:level=7
```

If automatic inlining is already enabled by default and you want to specify an inlining level (For example: 7), you enter:

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

Related information

- “-g” on page 103
- “-qipa” on page 181
- “Interprocedural analysis” in the *XL Fortran Optimization and Programming Guide*
- The `-qinline` inlining option in the *XL Fortran Optimization and Programming Guide*
- For a list of deprecated compiler options, see [Deprecated options](#)

-qintlog

Category

Floating-point and integer control

Purpose

Specifies that you can mix integer and logical data entities in expressions and statements.

Syntax

```
►► -q 

|          |
|----------|
| nointlog |
| intlog   |

 ◄◄
```

@PROCESS:

@PROCESS INTLOG | **NOINTLOG**

Defaults

-qnointlog

Usage

When **-qintlog** is specified, 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.

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

You can also use the intrinsic functions **IAND**, **IOR**, **IEOR**, and **NOT** to perform bitwise logical operations.

The **MOVE_ALLOC** intrinsic function cannot take one integer and one logical argument.

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) + 1
END
```

Related information

- **-qport=clogicals** option.

-qintsize

Category

Floating-point and integer control

Purpose

Sets the size of default **INTEGER** and **LOGICAL** data entities that have no length or kind specified.

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.

Syntax



@PROCESS:

@PROCESS INTSIZE(*bytes*)

Defaults

`-qintsize=4`

Parameters

bytes

Allowed sizes are 2, 4, or 8.

Usage

This option is intended to allow you to port programs unchanged from systems that have different default sizes for data. For example, you might need `-qintsize=2` for programs that are written for a 16-bit microprocessor or `-qintsize=8` for programs that are written for a CRAY computer. The default value of 4 for this option is suitable for code that is written specifically for many 32-bit computers. Note that specifying the `-q64` compiler option does not affect the default setting for `-qintsize`.

You might need to add **PARAMETER** statements to give explicit lengths to constants that you pass as arguments.

The specified size¹ applies to these data entities:

- **INTEGER** and **LOGICAL** specification statements with no length or kind specified.
- **FUNCTION** statements with no length or kind specified.
- Intrinsic functions that accept or return default **INTEGER** or **LOGICAL** arguments or return values unless you specify a length or kind in an **INTRINSIC** statement. Any specified length or kind must agree with the default size of the return value.
- Variables that are implicit integers or logicals.
- Integer and logical literal constants with no kind specified. If the value is too large to be represented by the number of bytes that you specified, the compiler chooses a size that is large enough. The range for 2-byte integers is $-(2^{15})$ to $2^{15}-1$, for 4-byte integers is $-(2^{31})$ to $2^{31}-1$, and for 8-byte integers is $-(2^{63})$ to $2^{63}-1$.
- Typeless constants in integer or logical contexts.
- In addition to types **INTEGER** and **LOGICAL**, `-qintsize` also works for **vector(integer)**. Specifying `-qintsize=2` is equivalent to specifying **vector(integer*2)**. Similarly, specifying `-qintsize=4` is equivalent to specifying **vector(integer*4)**. Specifying `-qintsize=8` is equivalent to **vector(integer*8)**.

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

- “-qrealize” on page 223
- *Type declaration: type parameters and specifiers* in the *XL Fortran Language Reference*

-qipa

Category

Optimization and tuning

@PROCESS

None.

Purpose

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

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

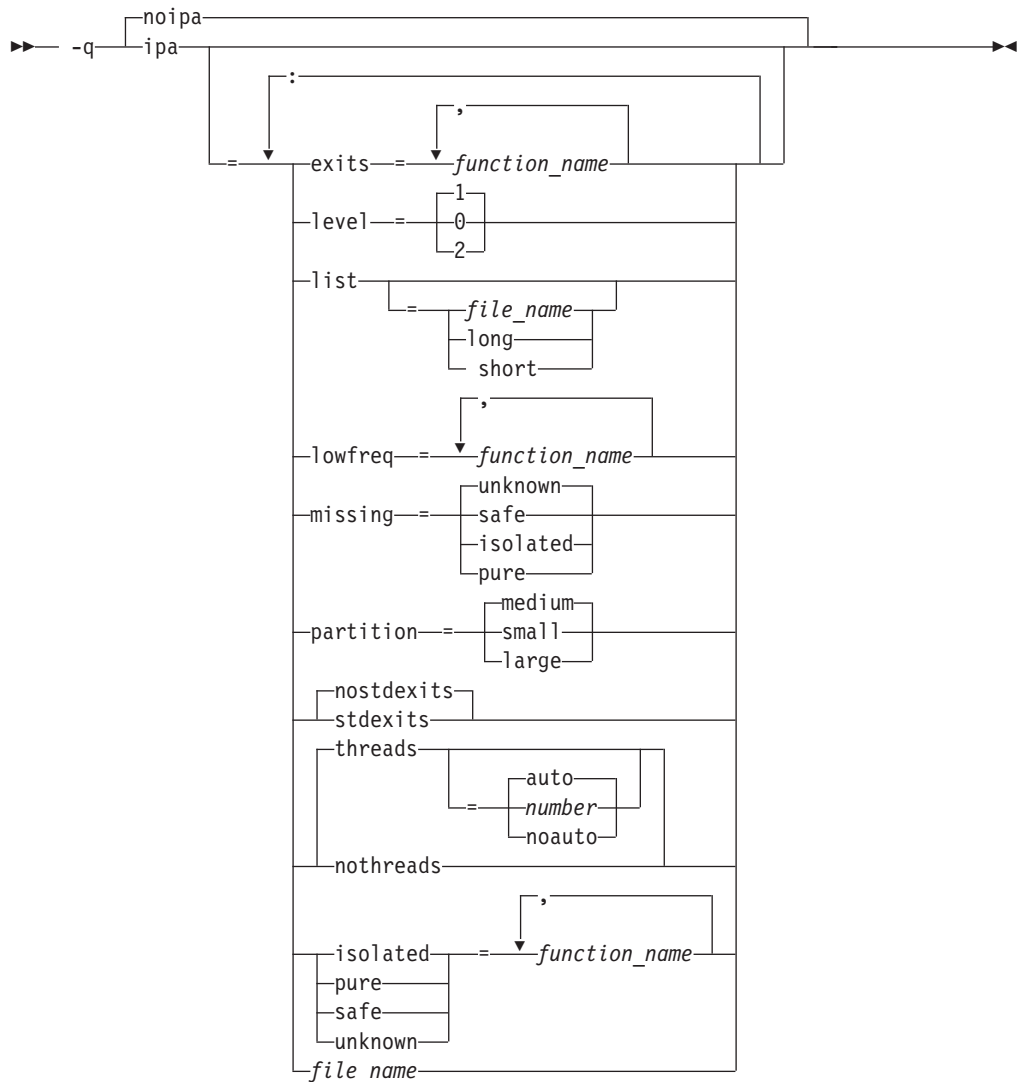
You can use **-qipa** during the compilation step, the link step, or both. If you compile and link in a single compiler invocation, only the link-time suboptions are relevant. If you compile and link in separate compiler invocations, only the compile-time suboptions are relevant during the compile step,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

Syntax

-qipa compile-time syntax



-qipa link-time syntax



Defaults

- **-qnoipa**

Parameters

The following are parameters that may be specified during a separate compile step only:

object | **noobject**

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

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

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

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

The following are parameters that may be specified during a combined compile and link in the same compiler invocation, or during a separate link step only:

exits

Specifies names of procedures which represent program exits. Program exits are calls which can never return and can never call any procedure which has been compiled with IPA pass 1. 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**.

isolated

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

Specifies the optimization level for interprocedural analysis. Valid suboptions are one of the following:

- 0** Performs only minimal interprocedural analysis and optimization.
- 1** Enables inlining, limited alias analysis, and limited call-site tailoring.
- 2** Performs full interprocedural data flow and alias analysis.

If you do not specify a level, the default is 1.

To generate data reorganization information, specify the optimization level **-qipa=level=2** or **-O5** together with **-qreport**. During the IPA link phase, the data reorganization messages for program variable data are written to the data reorganization section of the listing file. Reorganizations include common block splitting, array splitting, array transposing, memory allocation merging, array interleaving, and array coalescing.

list

Specifies that a listing file be generated during the link phase. The listing file contains information about transformations and analyses performed by IPA, as well as an optional object listing for each partition.

If you do not specify a *list_file_name*, the listing file name defaults to a.lst. If you specify **-qipa=list** together with any other option that generates a listing file, IPA generates an a.lst file that overwrites any existing a.lst file. If you have

a source file named a.f, the IPA listing will overwrite the regular compiler listing a.lst. You can use the **-qipa=list=list_file_name** suboption to specify an alternative listing file name.

Additional suboptions are one of the following:

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

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

lowfreq

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

missing

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.

Valid suboptions are one of the following:

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

isolated

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

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

unknown

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

The default is to assume **unknown**.

partition

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

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

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.

pure

Specifies *pure* procedures that are not compiled with **-qipa**. Any procedure

specified as *pure* must be *isolated* and *safe*, and must not alter the internal state or have side-effects, defined as potentially altering any data visible to the caller.

safe

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

stdexits | nostdexits

Specifies that certain predefined routines can be optimized as with the **exits** suboption. The procedures are: `abort`, `exit`, `_exit`, and `_assert`.

threads | nothreads

Runs portions of the IPA optimization process during pass 2 in parallel threads, which can speed up the compilation process on multi-processor systems. Valid suboptions for the **threads** suboption are as follows:

auto | noauto

When **auto** is in effect, the compiler selects a number of threads heuristically based on machine load. When **noauto** is in effect, the compiler spawns one thread per machine processor.

number

Instructs the compiler to use a specific number of threads. *number* can be any integer value in the range of 1 to 32 767. However, *number* is effectively limited to the number of processors available on your system.

Specifying **threads** with no suboptions implies **-qipa=threads=auto**.

unknown

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

file_name

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

The file format is the following:

```
# ... comment
attribute{, attribute} = name{, name}

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

where *attribute* is one of:

- exits
- lowfreq
- unknown
- safe
- isolated
- pure

Note:

- **-qipa=inline** and all of its associated suboptions are deprecated, **-qinline** replaces them all. For details, see `-qinline` and “Deprecated options” on page 92.

- As of the V11.1 release of the compiler, the **pdfname** suboption is deprecated; you should use **-qpdf1=pdfname** or **-qpdf2=pdfname** in your new applications. See “-qpdf1, -qpdf2” on page 208 for details.

Usage

Specifying **-qipa** automatically sets the optimization level to **-O2**. For additional performance benefits, you can also specify the **-qinline** option. The **-qipa** option extends the area that is examined during optimization and inlining from a single procedure to multiple procedures (possibly in different source files) and the linkage between them.

If any object file used in linking with **-qipa** was created with the **-qipa=noobject** option, any file containing an entry point (the main program for an executable program, or an exported function for a library) must be compiled with **-qipa**.

You can link objects created with different releases of the compiler, but you must ensure that you use a linker that is at least at the same release level as the newer of the compilers used to create the objects being linked.

Some symbols which are clearly referenced or set in the source code may be optimized away by IPA, and may be lost to **debug** or **nm** outputs. Using IPA together with the **-g** compiler will usually result in non-steppable output.

Note that if you specify **-qipa** with **-#**, the compiler does not display linker information subsequent to the IPA link step.

For recommended procedures for using **-qipa**, see "Optimizing your applications" in the *XL Fortran Optimization and Programming Guide*.

Examples

The following example shows how you might compile a set of files with interprocedural analysis:

```
xlf -c *.f -qip
xlf -o product *.o -qipa
```

Here is how you might compile the same set of files, improving the optimization of the second compilation, and the speed of the first compile step. Assume that there exist a set of routines, `user_trace1`, `user_trace2`, and `user_trace3`, which are rarely executed, and the routine `user_abort` that exits the program:

```
xlf95 -c *.f -qipa=noobject
xlf95 -c *.o -qipa=lowfreq=user_trace[123]:exit=user_abort
```

Related information

- **-qinline**
- “-qlibmpi” on page 190
- “-qpdf1, -qpdf2” on page 208
- “-S” on page 279
- Deprecated options
- Correct settings for environment variables

-qkeepparm

Category

Error checking and debugging

@PROCESS

None.

Purpose

When used with **-O2** or higher optimization, specifies whether function parameters are stored on the stack.

A procedure usually stores its incoming parameters on the stack at the entry point. However, when you compile code with optimization options enabled, the optimizer may remove these parameters from the stack if it sees an optimizing advantage in doing so.

Syntax

►► -q nokeepparm
keepparm ►►

Defaults

-qnokeepparm

Usage

When **-qkeepparm** is in effect, parameters are stored on the stack even when optimization is enabled. This option then provides access to the values of incoming parameters to tools, such as debuggers, simply by preserving those values on the stack. However, this may negatively impact execution performance.

When **-qnokeepparm** is in effect, parameters are removed from the stack if this provides an optimization advantage.

-qlanglvl

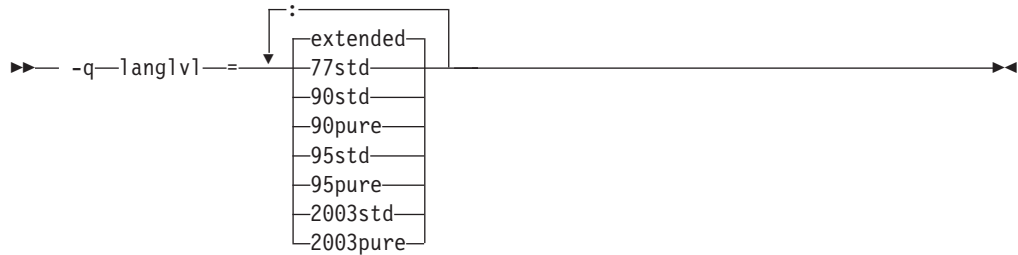
Category

Language element control

Purpose

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.

Syntax



@PROCESS:

@PROCESS LANGLVL({*suboption*})

Defaults

`-qlanglvl=extended`

Parameters

77std Accepts the language that the ANSI FORTRAN 77 standard specifies and reports anything else using language-level messages.

90std Accepts the language that the ISO Fortran 90 standard specifies and reports anything else using language-level messages.

90pure

The same as **90std** except that it also issues language-level messages for any obsolescent Fortran 90 features used.

95std Accepts the language that the ISO Fortran 95 standard specifies and reports anything else using language-level messages.

95pure

The same as **95std** except that it also issues language-level messages for any obsolescent Fortran 95 features used.

2003std

Accepts the language that the ISO Fortran 95 standard specifies, as well as all Fortran 2003 features supported by XL Fortran, and reports anything else using language-level messages.

2003pure

The same as **2003std** except that it also issues language-level messages for any obsolescent Fortran 2003 features used.

extended

Accepts the full Fortran 95 language standard, all Fortran 2003 features supported by XL Fortran, and all extensions, effectively turning off language-level checking.

Usage

When a `-qlanglvl` setting is specified, the compiler issues a message with severity code **L** if it detects syntax that is not allowed by the language level that you specified.

The `-qflag` option can override the `-qlanglvl` option.

The **langlvl** runtime option, which is described in “Setting runtime options” on page 36, helps to locate runtime extensions that cannot be checked for at compile time.

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 table shows examples of how some **-qlanglvl** suboptions affect this sample program:

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

Related information

- “-qflag” on page 151
- “-qhalt” on page 166
- “-qsa” on page 228
- **langlvl** runtime option in “Setting runtime options” on page 36

-qlibansi

Category

Optimization and tuning

@PROCESS

None.

Purpose

Assumes that all functions with the name of an ANSI C library function are, in fact, the library functions and not a user function with different semantics.

Syntax

►► -q no libansi libansi _____►►

Defaults

-qnolibansi

Usage

This option will allow the optimizer to generate better code because it will know about the behavior of a given function, such as whether or not it has any side effects.

Note: Do not use this option if your application contains your own version of a library function that is incompatible with the standard one.

Related information

See “-qipa” on page 181.

-qlibmpi

Category

“Optimization and tuning” on page 86

@PROCESS

None

Purpose

Asserts that all functions with Message Passing Interface (MPI) names are in fact MPI functions and not a user function with different semantics.

Syntax

►► -q no libmpi libmpi _____►►

Defaults

-qnolibmpi

Usage

MPI is a library interface specification for message passing. It addresses the message-passing parallel programming model in which data is moved from the address space of one process to another through cooperative operations. For details about MPI, see the Message Passing Interface Forum.

-qlibmpi allows the compiler to generate better code because it knows about the behavior of a given function, such as whether or not it has any side effects.

When you use **-qlibmpi**, the compiler assumes that all functions with the name of an MPI library function are in fact MPI functions. **-qnolibmpi** makes no such assumptions.

Note: You cannot use this option if your application contains your own version of the library function that is incompatible with the standard one.

Examples

To compile `myprogram.f`, enter the following command:

```
xlf -O5 myprogram.f -qlibmpi
```

Related information

- Message Passing Interface Forum
- “-qipa” on page 181

-qlinedebug

Category

Error checking and debugging

Purpose

Generates only line number and source file name information for a debugger.

When **-qlinedebug** is in effect, the compiler produces minimal debugging information, so the resulting object size is smaller than that produced by the **-g** debugging option. You can use the debugger to step through the source code, but you will not be able to see or query variable information. The traceback table, if generated, will include line numbers.

Syntax

```
→ -q [no l i n e d e b u g] [l i n e d e b u g] →
```

@PROCESS:

@PROCESS `LINEDebug` | `NOLINEDEBUG`

Defaults

`-qnolinedebug`

Usage

When **-qlinedebug** is in effect, function inlining is disabled.

As with all debug information, the output of **-qlinedebug** may be incomplete or misleading if the code is optimized.

The **-g** option overrides the **-qlinedebug** option. If you specify **-g** with **-qnolinedebug** on the command line, **-qlinedebug** is ignored and a warning is issued.

Related information

- “-g” on page 103
- “-O” on page 108

-qlist

Category

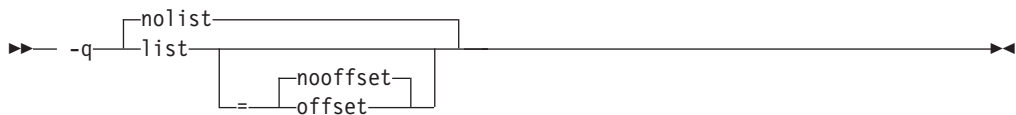
Listings, messages, and compiler information

Purpose

Produces a compiler listing file that includes an object listing.

When **-qlist** is in effect, a listing file is generated with a `.lst` suffix for each source file named on the command line.

Syntax



@PROCESS:

@PROCESS LIST[[([NO]OFFSET)] | **NOLIST**

Defaults

-qnolist

Parameters

offset | **nooffset**

When **-qlist=offset** is in effect, the listing will show the offset from the start of the procedure rather than the offset from the start of code generation. This suboption allows any program reading the `.lst` file to add the value of the PDEF and the line in question, and come up with the same value whether **offset** or **nooffset** is specified.

The **offset** suboption is relevant only if there are multiple procedures in a compilation unit. For example, this may occur if nested procedures are used in a program.

Specifying **-qlist** with no suboption is equivalent to **-qlist=nooffset**.

Usage

You can use the object listing to help understand the performance characteristics of the generated code and to diagnose execution problems.

If you specify `-qipa` and want to generate the IPA listing file, use the `-qipa=list=filename` suboption to get an alternative listing.

The `-qnoprint` compiler option overrides this option.

Related information

- “Listings, messages, and compiler information” on page 85
- “Object section” on page 306
- “-qnoprint” on page 203
- “-S” on page 279
- *Program units and procedures* in the *XL Fortran Language Reference*

-qlistfmt

Category

Listings, messages, and compiler information

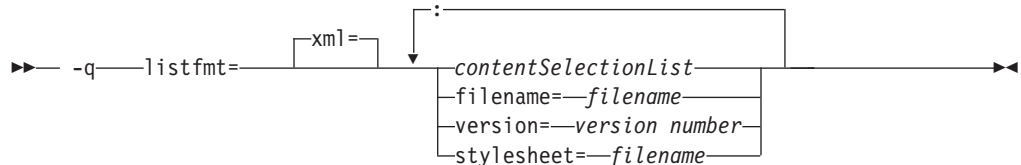
@PROCESS

None.

Purpose

Creates an XML report to assist with finding optimization opportunities.

Syntax



Defaults

This option is not on by default. If no `contentSelectionList` options are selected in their positive form, no report is produced. The default is `-qlistfmt=xml=none`.

Parameters

The following list describes `-qlistfmt` parameters:

xml

Indicates that the report should be generated in XML format.

contentSelectionList

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

data | nodata

Produces data reorganization information.

inlines | noinlines

Produces inlining information.

pdf | nopdf

Produce profile-directed feedback information.

transforms | nottransforms

Produces loop transformation information.

all Produces all available report information.

none

Does not produce a report.

filename

Specifies the name of the report file. One file is produced during the compile phase, and one file is produced during the IPA link phase. If no filename is specified, a file with the suffix `.xml` is generated in a way that is consistent with the rules of name generation for the given platform. For example, if compiling `foo.f` the generated XML files are `foo.xml` from the compile step and `a.xml` from the link step.

Note: If you compile and link in one step and use this suboption to specify a file name for the report, the information from the IPA link step will overwrite the information generated during the compile step.

The same will be true if you compile multiple files using the `filename` suboption. The compiler creates an report for each file so the report of the last file compiled will overwrite the previous reports. For example,

```
xlf -qlistfmt=xml=all:filename=abc.xml -O3 myfile1.f myfile2.f myfile3.f
```

will result in only one report, `abc.xml` based on the compilation of the last file `myfile3.f`

stylesheet

Specifies the name of an existing XML stylesheet for which an `xml-stylesheet` directive is embedded in the resulting report. The default behavior is to not include a stylesheet. The stylesheet shipped with XL Fortran is `x1style.xml`. This stylesheet renders the XML to an easily read format when viewed using a browser that supports XSLT.

To view the XML report created with the **stylesheet** suboption, you must place the actual stylesheet (`x1style.xml`) and the XML message catalogue (`XMLMessages-lang.xml` where *lang* refers to the locale set on the compilation machine) in the path specified by the **stylesheet** suboption. For example, if `a.xml` is generated with **stylesheet=x1style.xml**, `x1style.xml` and `XMLMessages-lang.xml` must be placed into the same directory as `a.xml`, before you can properly view `a.xml` with a browser. The message catalogs and stylesheet are installed in the `/opt/ibmcomp/xlf/13.1/listings/` directory.

version

If you have written a tool that requires a certain version of this report, you should specify the version. IBM XL Fortran for Linux, V13.1 creates reports at `v1.0` so if you have written a tool to consume these reports, you should specify `version=v1.0`.

Usage

The information produced in the report by the **-qlistfmt** option depends on which optimization options are used to compile the program.

- When used with an option that enables inlining such as **-qinline**, the report shows which functions were inlined and why others were not inlined.
- When used with an option that enables loop unrolling, the report contains a summary of how program loops are optimized. The report also includes diagnostic information to show why specific loops cannot be vectorized. For **-qlistfmt** to generate information about loop transformations, you must also specify at least one of the following options:
 - **-qsimd=auto**
 - **-qsmp**
 - **-O5**
 - **-qipa=level=2**
- When used with an option that enables parallel transformations, the report contains information about parallel transformations. For **-qlistfmt** to generate information about parallel transformations or parallel performance messages, you must also specify at least one of the following options:
 - **-qsmp**
 - **-O5**
 - **-qipa=level=2**
- When used with the option that enables profiling, **-qpdf**, the report contains information about call and block counts and cache misses.
- When used with an option that produces data reorganizations such as **-qipa=level=2**, the report contains information about those reorganizations.

If no *contentSelectionList* options are selected in their positive form, no report is produced.

Examples

If you want to compile `myprogram.f` to produce an XML 1.0 report that shows how loops are optimized, enter:

```
xlf -qhot -O3 -qlistfmt=xml=transforms myprogram.f
```

If you want to compile `myprogram.f` to have the XML report show which functions were inlined, enter:

```
xlf -qinline -qlistfmt=xml=inlines myprogram.f
```

Related information

- “-qreport” on page 226
- “Using compiler reports to diagnose optimization opportunities” in the *XL Fortran Optimization and Programming Guide*

-qlistopt

Category

Listings, messages, and compiler information

Purpose

Produces a compiler listing file that includes all options in effect at the time of compiler invocation.

When **listopt** is in effect, a listing file is generated with a `.lst` suffix for each source file named on the command line. The listing shows options in effect as set by the compiler defaults, the configuration file, and command line settings.

Syntax

►► `-q` no listopt
listopt ◀◀

@PROCESS:

@PROCESS LISTOPT | NOLISTOPT

Defaults

`-qnolistopt`

Usage

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.

Options that are always displayed in the listing include:

- 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

The **-qnoprint** compiler option overrides this option.

Related information

- “Listings, messages, and compiler information” on page 85
- “Options section” on page 301

-qlog4

Category

Portability and migration

Purpose

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.

Syntax

►► -q no log4
log4 —————►►

@PROCESS:

@PROCESS LOG4 | NOLOG4

Defaults

-qno**log4**, which makes the result depend on the lengths of the operands.

Usage

Specifying -q**log4** makes the result a **LOGICAL(4)**.

If you use -q**intsize** to change the default size of logicals, -q**log4** is ignored.

-qmaxmem

Category

Optimization and tuning

Purpose

Limits the amount of memory that the compiler allocates while performing specific, memory-intensive optimizations to the specified number of kilobytes.

Syntax

►► -q maxmem = *Kbytes* —————►►

@PROCESS:

@PROCESS MAXMEM(*Kbytes*)

Defaults

- **maxmem=8192** when **-O2** is in effect.
- **maxmem=-1** when **-O3** or higher optimization is in effect.

Parameters

Kbytes

The number of kilobytes worth of memory to be used by optimizations. The limit is the amount of memory for specific optimizations, and not for the compiler as a whole. Tables required during the entire compilation process are not affected by or included in this limit.

A value of **-1** permits each optimization to take as much memory as it needs without checking for limits.

Usage

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 8192 or higher, and decrease it if the compilation continues to require too much storage.

Note:

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 before they exceed the maximum available address space, but not before they 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

- “**-O**” on page 108
- *Optimizing your applications* in the *XL Fortran Optimization and Programming Guide*

-qmbcs

Category

Language element control

Purpose

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.

Syntax

►► -q nombcs
mbsc _____ ►►

@PROCESS:

@PROCESS MBCS | NOMBCS

Defaults

-qnombcs

Usage

Each byte of a multibyte character is counted as a column.

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.

To read or write Unicode data, set the locale value to **UNIVERSAL** at run time. If you do not set the locale, you might not be able to interchange data with Unicode-enabled applications.

-qminimaltoc

Category

Optimization and tuning

@PROCESS

None.

Purpose

In 64-bit compilation mode, minimizes the number of entries in the global entity table of contents (TOC).

Syntax

►► -q nominaltoc
minimaltoc _____ ►►

Defaults

-qnominaltoc

Usage

This compiler option applies to 64-bit compilations only.

By default, the compiler will allocate at least one TOC entry for each unique, non-automatic variable reference in your program. Currently, only 8192 TOC entries are available and duplicate entries are not discarded. This can cause errors when linking large programs in 64-bit mode if your program exceeds 8192 TOC entries.

Specifying **-qminimaltoc** ensures that the compiler creates only one TOC entry for each compilation unit. Specifying this option can minimize the use of available TOC entries, but its use impacts performance.

Use the **-qminimaltoc** option with discretion, particularly with files that contain frequently executed code.

-qmixed

Category

Input control

Purpose

This is the long form of the “-U” on page 281 option.

Syntax

►► -q nomixed
mixed ◀◀

@PROCESS:

@PROCESS MIXED | NOMIXED

Defaults

-qnomixed

-qmkshrobj

Category

Output control

@PROCESS

None.

Purpose

Creates a shared object from generated object files.

Use this option, together with the related options described later in this topic, instead of calling the linker directly to create a shared object. The advantage of using this option is that it is compatible with **-qipa** link-time optimizations (such as those performed at **-O5**).

Syntax

►► **-qmkshrobj** ◀◀

Defaults

By default, the output object is linked with the runtime libraries and startup routines to create an executable file.

Usage

Specifying **-qmkshrobj** implies **-qplic**.

You can also use the following related options with **-qmkshrobj**:

-o *shared_file*

The name of the file that holds the shared file information. The default is `a.out`.

-e *name*

Sets the entry name for the shared executable to *name*.

Note: Options **-qmkshrobj** and **-qstaticlink** are incompatible and cannot be specified together. If you specify **-qmkshrobj** and **-qstaticlink** (or **-qstaticlink=libgcc**) together, the driver issues the following message: 1501–264 The options \$1 and \$2 are incompatible. Option \$1 is ignored.

For detailed information about using **-qmkshrobj** to create shared libraries, see “Compiling and linking a library” on page 24.

Examples

To construct the shared library `big_lib.so` from three smaller object files, enter the following command:

```
xlf -qmkshrobj -o big_lib.so lib_a.o lib_b.o lib_c.o
```

Related information

- “**-e**” on page 101
- “**-qipa**” on page 181
- “**-o**” on page 110
- “**-qplic**” on page 215

-qmoddir

Category

Output control

@PROCESS

None.

Purpose

Specifies the location for any module (**.mod**) files that the compiler writes.

Syntax

►► -q—moddir=*directory*◄◄

Defaults

Not applicable.

Usage

If you do not specify **-qmoddir**, the **.mod** files are placed in the current directory.

To read the **.mod** files from this directory when compiling files that reference the modules, use the **-I** option.

Related information

- “XL Fortran output files” on page 27
- “-I” on page 104
- *Modules* section in the *XL Fortran Language Reference*.

-qmodule

Category

Portability and migration

@PROCESS

None.

Purpose

Specifies that the compiler should use the XL Fortran Version 8.1 naming convention for non-intrinsic module files.

Syntax

►► -q—module=

nomangle81
mangle81

◄◄

Defaults

-qmodule=nomangle81

Usage

This option allows you to produce modules and their associated object files with the V13.1 compiler and link these object files with others compiled with the Version 8.1 compiler.

Use this option only if you need to link applications that were compiled with the Version 8.1 compiler.

It is recommended that you avoid using this compiler option and recompile old code and modules with the new version of the compiler, if possible. Doing so will avoid any naming conflicts between your modules and intrinsic modules.

Related information

- *Modules* section in the *XL Fortran Language Reference*.
- *Conventions for XL Fortran external names* in the *XL Fortran Optimization and Programming Guide*
- “Avoiding naming conflicts during linking” on page 34

-qnoprint

Category

Listings, messages, and compiler information

@PROCESS

None.

Purpose

Prevents the compiler from creating the listing file, regardless of the settings of other listing options.

Syntax

▶— -q—noprint—▶

Defaults

Not applicable.

Usage

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.

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

- “Listings, messages, and compiler information” on page 85

-qnullterm

Category

Language element control

Purpose

Appends a null character to each character constant expression that is passed as a dummy argument, making 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.

Syntax

►► -q nonullterm
nullterm ◄◄

@PROCESS:

@PROCESS NULLTERM | NONULLTERM

Defaults

-qnonullterm

Usage

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
- 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* in the *XL Fortran Optimization and Programming Guide*.

-qobject

Category

Error checking and debugging

Purpose

Specifies whether to produce an object file or to stop immediately after checking the syntax of the source files.

Syntax

►► -q  ◀◀

@PROCESS:

@PROCESS Object | NOObject

Defaults

-qobject

Usage

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.

The **-qhalt** option can override the **-qobject** option, and **-qnoobject** can override **-qhalt**.

Related information

- “Listings, messages, and compiler information” on page 85
- “Object section” on page 306
- The compiler phases in the *Getting Started with XL Fortran*.
- “-qhalt” on page 166

-qonetrip

Category

Language element control

Purpose

This is the long form of the **-1** option.

Syntax

►► — -q —

noonetrip
onetrip

 —►►

@PROCESS:

@PROCESS ONETRIP | NOONETRIP

Defaults

-qnoonetrip

-qoptdebug

Category

Error checking and debugging

@PROCESS

None.

Purpose

When used with high levels of optimization, produces files containing optimized pseudocode that can be read by a debugger.

An output file with a **.optdbg** extension is created for each source file compiled with **-qoptdebug**. You can use the information contained in this file to help you understand how your code actually behaves under optimization.

Syntax

►► -q nooptdebug
optdebug ◀◀

Defaults

-qnooptdebug

Usage

-qoptdebug only has an effect when used with an option that enables the high-level optimizer, namely **-O3** or higher optimization level, or **-qhot**, **-qsmp**, **-qipa**, or **-qpdf**. You can use the option on both compilation and link steps. If you specify it on the compile step, one output file is generated for each source file. If you specify it on the **-qipa** link step, a single output file is generated.

You must still use the **-g** or **-qlinedebug** option to include debugging information that can be used by a debugger.

For more information and examples of using this option, see "Using -qoptdebug to help debug optimized programs" in the *XL Fortran Optimization and Programming Guide*.

Related information

- “-O” on page 108
- “-qhot” on page 167
- “-qipa” on page 181
- “-qpdf1, -qpdf2” on page 208
- “-qsmp” on page 237
- “-g” on page 103
- “-qlinedebug” on page 191

-qoptimize

Purpose

This is the long form of the -O option.

Syntax

►► -q NOOPTimize
OPTimize==level ◀◀

@PROCESS:

@PROCESS OPTimize[(level)] | NOOPTimize

Defaults

-qnooptimize

-qpdf1, -qpdf2

Category

Optimization and tuning

@PROCESS

None.

Purpose

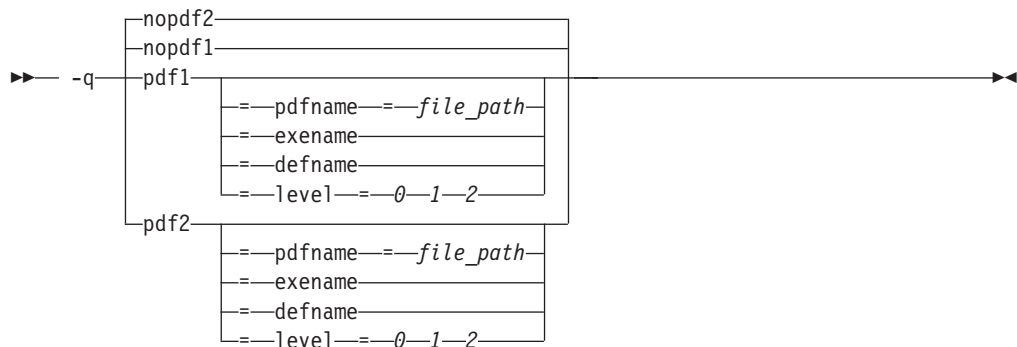
Tunes optimizations through *profile-directed feedback* (PDF), where results from sample program execution are used to improve optimization near conditional branches and in frequently executed code sections.

PDF is a two-step process. You first compile the application with **-qpdf1** and a minimum optimization level of **-O2**, with linking. You then run the resulting application with a typical data set. During the test run, profile data is written to a profile file. By default, the profile file is named `._pdf` and is saved in the current working directory, or in the directory named by the `PDFDIR` environment variable, if it is set. You then recompile and link or relink the application with **-qpdf2** and an optimization level used for **-qpdf1**, which fine-tunes the optimizations applied according to the profile data collected during the program execution.

You can use old profiling information. In previous releases, when you modify the source file or compiler options and compile with **-qpdf2**, the compilation stops with an error. As of IBM XL Fortran for Linux, V13.1, you see a list of warnings but compilation does not stop. However, using different compiler options between different stages of PDF does not give you any benefits for using PDF.

PDF is intended to be used after other debugging and tuning is finished, as one of the last steps before putting the application into production.

Syntax



Defaults

-qnopdf1, -qnopdf2

Parameters

defname

Reverts the PDF file to its default file name.

exename

Generates the name of the PDF file based on what you specify with the **-o** option. For example, you can use **-qpdf1=exename -o foo foo.f** to generate a PDF file called **.foo_pdf**.

level=0 | 1 | 2

Supports multiple-pass profiling, block counter, call counter and extended value profiling. You can compile your application with **-qpdf1=level=0|1|2** to generate profiling data with different levels of optimization. Note that **-qpdf1=level=0** and **-qpdf1=level=1** support single-pass profiling, whereas **-qpdf1=level=2** supports multiple-pass profiling. The following is a list of detailed descriptions for each level of optimization:

- **0** is the basic compiler instrumentation that generates lower overhead than **-qpdf1=level=1**.
- **1** is the default compiler instrumentation that is equivalent to **-qpdf1** in previous releases.
- **2** is a more aggressive compiler instrumentation. Cache-miss profiling is enabled on **pSeries®** in addition to basic block counter and value profiling performed at **-qpdf1=level=1**. This suboption is supported at all optimization levels where PDF is enabled. You can use **-qpdf1=level=2** to aggressively gather profile information, execute applications with typical input data, and use **-qpdf2** to optimize the executable.

pdfname= *file_path*

Specifies the path to the file that will hold the profile data. By default, the file name is **.pdf**, and it is placed in the current working directory or in the directory named by the **PDFDIR** environment variable. You can use the **pdfname** suboption to allow you to do simultaneous runs of multiple executables using the same PDF directory. This is especially useful when tuning with PDF on dynamic libraries.

Usage

You must compile the main program with PDF for profiling information to be collected at run time.

If you do not want the optimized object files to be relinked during the second step, specify **-qpdf2 -qnoipa**.

If you want to specify an alternate path and file name for the profile file, use the **pdfname** suboption. Alternatively, you can use the **PDFDIR** environment variable to specify the absolute path name for the directory. To generate the name of the PDF files you specify, you can use the **exename** suboption. To revert the PDF file to its default name, use the **defname** suboption. Do not compile or run two different applications that use the same profiling directory at the same time, unless you have used the **pdfname** suboption to distinguish the sets of profiling information.

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

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

-qshowpdf

Provides additional information, such as block and function call counts, to the profile file. See “-qshowpdf ” on page 233 for more information.

For recommended procedures of using PDF, see "Profile-directed feedback" in the *XL Fortran Optimization and Programming Guide*.

The following utility programs, found in `/opt/ibmcomp/xf/13.1/bin/`, are available for managing the directory to which profile data is written:

cleanpdf

►► cleanpdf [directory_path]

Removes all profiling information from the directory specified by *directory_path*; or if *pathname* is not specified, from the directory set by the PDFDIR environment variable; or if PDFDIR is not set, from the current directory. Removing profiling information reduces runtime overhead if you change the program and then go through the PDF process again.

Run **cleanpdf** only when you are finished with the PDF process for a particular application. Otherwise, if you want to resume using PDF with that application, you will need to recompile all of the files again with **-qpdf1**.

mergepdf

►► mergepdf [-r scaling] input -o output [-n] [-v]

Merges two or more PDF records into a single PDF output record.

-r scaling

Specifies the scaling ratio for the PDF record file. This value must be greater than zero and can be either an integer or floating point value. If not specified, a ratio of 1.0 is assumed.

input Specifies the name of a PDF input record file, or a directory that contains PDF record files.

-o output

Specifies the name of the PDF output record file, or a directory to which the merged output will be written.

-n If specified, PDF record files are not normalized. If not specified, **mergepdf** normalizes records based on an internally-calculated ratio before applying any user-defined scaling factor.

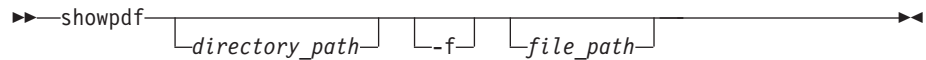
-v Specifies verbose mode, and causes internal and user-specified scaling ratios to be displayed to standard output.

resetpdf

►► resetpdf [directory_path]

Same as **cleanpdf**, described above.

showpdf



Displays the function call and block counts written to the profile file, specified by the **-f** option, during a program run. To use this command, you must first compile your application specifying both **-qpdf1** and **-qshowpdf** compiler options on the command line.

Predefined macros

None.

Examples

Here is a simple example:

```
# Compile all files with -qpdf1.
xlf -qpdf1 -O3 file1.f file2.f file3.f

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

# Recompile all files with -qpdf2.
xlf -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.
xlf -qpdf1 -O3 -c file1.f file2.f file3.f

# This file is not so important to optimize.
xlf -c file4.f

# Non-PDF object files such as file4.o can be linked in.
xlf -qpdf1 -O3 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

# No need to recompile the source of non-PDF object files (file4.f).
xlf -qpdf2 -O3 file1.f file2.f file3.f

# Link all the object files into the final application. */
xlf -qpdf2 -O3 file1.o file2.o file3.o file4.o
```

Here is an example that bypasses recompiling the source with **-qpdf2**:

```
# Compile source with -qpdf1.
xlf -O3 -qpdf1 -c file.f

# Link in object file.
xlf -O3 -qpdf1 file.o
```

```

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

# Link in object file from qpdf1 pass.
# (Bypass source recompilation with -qpdf2.)
xlf -O3 -qpdf2 file.o

```

Here is an example of using pdf1 and pdf2 objects:

```

# Compile source with -qpdf1.
xlf -c -qpdf1 -O3 file1.f file2.f

# Link in object files.
xlf -qpdf1 -O3 file1.o file2.o

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

# Link in the mix of pdf1 and pdf2 objects.
xlf -qpdf2 -O3 file1.o file2.o

```

Here is an example that creates PDF-optimized object files without relinking into an executable:

```

# Compile source with -qpdf1.
xlf -c -O3 -qpdf1 file1.f file2.f file3.f

# Link in object files.
xlf -O3 -qpdf1 file1.o file2.o file3.o

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

# Recompile the instrumented source files
xlf -c -O3 -qpdf2 -qnoipa file1.f file2.f file3.f

```

Here is an example that reduces possible runtime instrumentation overhead:

```

#Compile all files with -qpdf1=level=0.
xlf -qpdf1=level=0 -O3 file1.f file2.f file3.f

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

#Recompile all files with -qpdf2.
xlf -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 an example of multiple pass profiling:

```

#Compile all files with -qpdf1=level=2.
xlf -qpdf1=level=2 -O5 file1.f file2.f file3.

#Run with one set of input data, the profiling information is recorded
#in ._pdf by default.
./a.out < sample.data

#Recompile all files with -qpdf1=level=2 again.
#The compiler will read the previous profile data,
#refine instrumentation and generate a new instrumented executable.

xlf -qpdf1=level=2 -O5 file1.f file2.f file3.f

#Run it again, the profiling information is recorded in
#._pdf.1

```

```

./a.out < sample.data

#Recompile all files with -qpdf2

xlf -qpdf2 -O5 file1.f file2.f file3.f

#The program should now run faster than
#without PDF if the sample data is typical.

Here is an example that uses -qpdf[1|2]=exename:
#Compile all files with -qpdf1=exename.
xlf -qpdf1=exename -O5 -o final file1.f file2.f file3.f

#Run executable with sample input data.

./final < typical.data

#List the content of the directory.
>ls -lrta

-rw-r--r-- 1 user staff 50 Dec 05 13:18 file1.f
-rw-r--r-- 1 user staff 50 Dec 05 13:18 file2.f
-rw-r--r-- 1 user staff 50 Dec 05 13:18 file3.f
-rwxr-xr-x 1 user staff 12243 Dec 05 17:00 final
-rwxr-Sr-- 1 user staff 762 Dec 05 17:03 .final_pdf

#Recompile all files with -qpdf2=exename.

xlf -qpdf2=exename -O5 -o final file1.f file2.f file3.f

#The program is now optimized using PDF information.

```

Related information

- “-qshowpdf ” on page 233
- “-qipa” on page 181
- -qprefetch
- “-qreport” on page 226
- “XL Fortran input files” on page 25
- “XL Fortran output files” on page 27
- “Profile-directed feedback” in the *XL Fortran Optimization and Programming Guide*
- Correct settings for environment variables

-qphsinfo

Category

Listings, messages, and compiler information

Purpose

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

Syntax

```

➔ -q nophsinfo  
phsinfo ➔

```

@PROCESS:

@PROCESS PHSINFO | **NOPHSINFO**

Defaults

-qnophsinfo

Usage

The output takes the form *number1/number2* for each phase where *number1* represents the CPU time used by the compiler and *number2* represents the total of the compiler time and the time that the CPU spends handling system calls.

Examples

To compile **app.f**, which consists of 3 compilation units, and report the time taken for each phase of the compilation, enter:

```
xlf90 app.f -qphsinfo
```

The output will look similar to:

```
FORTRAN phase 1 ftphas1      TIME = 0.000 / 0.000
** m_module   === End of Compilation 1 ===
FORTRAN phase 1 ftphas1      TIME = 0.000 / 0.000
** testassign === End of Compilation 2 ===
FORTRAN phase 1 ftphas1      TIME = 0.000 / 0.010
** dataassign === End of Compilation 3 ===
HOT           - Phase Ends; 0.000/ 0.000
HOT           - Phase Ends; 0.000/ 0.000
HOT           - Phase Ends; 0.000/ 0.000
W-TRANS      - Phase Ends; 0.000/ 0.010
OPTIMIZ      - Phase Ends; 0.000/ 0.000
REGALLO      - Phase Ends; 0.000/ 0.000
AS           - Phase Ends; 0.000/ 0.000
W-TRANS      - Phase Ends; 0.000/ 0.000
OPTIMIZ      - Phase Ends; 0.000/ 0.000
REGALLO      - Phase Ends; 0.000/ 0.000
AS           - Phase Ends; 0.000/ 0.000
W-TRANS      - Phase Ends; 0.000/ 0.000
OPTIMIZ      - Phase Ends; 0.000/ 0.000
REGALLO      - Phase Ends; 0.000/ 0.000
AS           - Phase Ends; 0.000/ 0.000
1501-510     Compilation successful for file app.f.
```

Each phase is invoked three times, corresponding to each compilation unit. FORTRAN represents front-end parsing and semantic analysis, HOT loop transformations, W-TRANS intermediate language translation, OPTIMIZ high-level optimization, REGALLO register allocation and low-level optimization, and AS final assembly.

Compile **app.f** at the **-O4** optimization level with **-qphsinfo** specified:

```
xlf90 myprogram.f -qphsinfo -O4
```

The following output results:

```
FORTRAN phase 1 ftphas1      TIME = 0.010 / 0.020
** m_module   === End of Compilation 1 ===
FORTRAN phase 1 ftphas1      TIME = 0.000 / 0.000
** testassign === End of Compilation 2 ===
FORTRAN phase 1 ftphas1      TIME = 0.000 / 0.000
** dataassign === End of Compilation 3 ===
HOT           - Phase Ends; 0.000/ 0.000
HOT           - Phase Ends; 0.000/ 0.000
HOT           - Phase Ends; 0.000/ 0.000
IPA          - Phase Ends; 0.080/ 0.100
1501-510     Compilation successful for file app.f.
IPA          - Phase Ends; 0.050/ 0.070
```

W-TRANS	- Phase Ends;	0.010/	0.030
OPTIMIZ	- Phase Ends;	0.020/	0.020
REGALLO	- Phase Ends;	0.040/	0.040
AS	- Phase Ends;	0.000/	0.000

Note that during the IPA (interprocedural analysis) optimization phases, the program has resulted in one compilation unit; that is, all procedures have been inlined.

Related information

.The compiler phases in the *Getting Started with XL Fortran*

-qpic

Category

Object code control

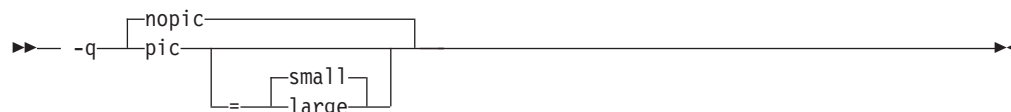
@PROCESS

None.

Purpose

Generates Position-Independent Code suitable for use in shared libraries.

Syntax



Defaults

- **-qnopic** in 32-bit compilation mode.
- **-qpic=small** in 64-bit compilation mode.

Parameters

small

Instructs the compiler to assume that the size of the Global Offset Table is no larger than 64 Kb.

large

Allows the Global Offset Table to be larger than 64 Kb in size, allowing more addresses to be stored in the table. Code generated with this option is usually larger than that generated with **-qpic=small**.

Specifying **-qpic** without any suboptions is equivalent to **-qpic=small**.

Usage

When **-q64** is in effect, **-qpic** is enabled and cannot be disabled.

Related information

- “-q32” on page 112
- “-q64” on page 113

-qport

Category

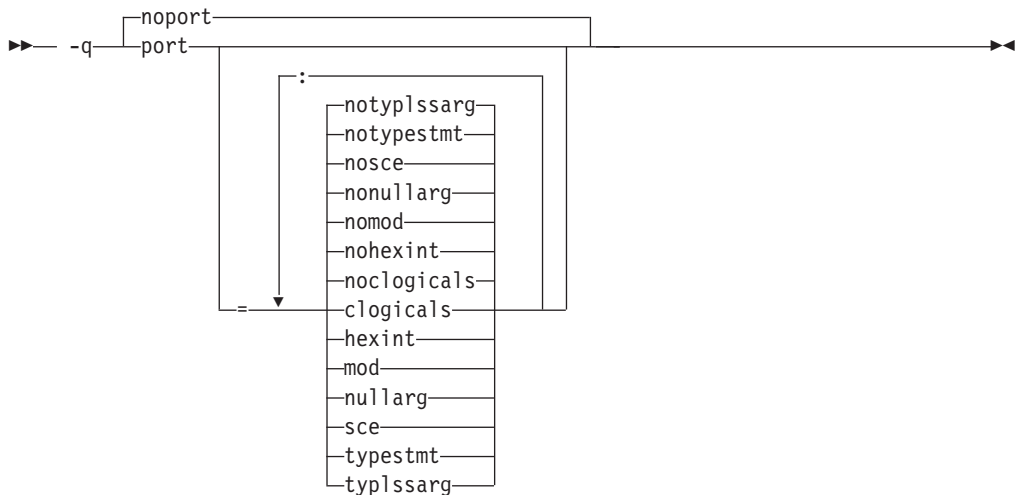
Portability and migration

Purpose

Provides options to accommodate other Fortran language extensions when porting programs to XL Fortran.

A particular **-qport** suboption will always function independently of other **-qport** and compiler options.

Syntax



@PROCESS:

@PROCESS PORT[(*suboptions*)] | **NOPORT**

Defaults

-qnoport

Parameters

cllogicals | **noclogicals**

When **cllogicals** is in effect, the compiler treats all non-zero integers that are used in logical expressions as TRUE. You must specify **-qintlog** for **-qport=cllogicals** to take effect.

The **-qport=cllogicals** option is useful when porting applications from other Fortran compilers that expect this behavior. However, it is unsafe to mix programs that use different settings for non-zero integers if they share or

pass logical data between them. Data files already written with the default **-qintlog** setting will produce unexpected behavior if read with the **-qport=clogicals** option active.

hexint | **nohexint**

When **hexint** is in effect, 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 **mod** relaxes existing constraints on the **MOD** intrinsic function, allowing two arguments of the same data type to be of different kind type parameters. The result will be of the same type as the argument, but with the larger kind type parameter value.

nullarg | **nonnullarg**

For an external or internal procedure reference, specifying **nullarg** causes the compiler to treat an empty argument, which is delimited by a left parenthesis and a comma, two commas, or a comma and a right parenthesis, as a null argument. This suboption has no effect if the argument list is empty.

Examples of empty arguments are:

```
call foo(,,z)
```

```
call foo(x,,z)
```

```
call foo(x,y,)
```

The following program includes a null argument.

Fortran program:

```
program nularg
real(4) res/0.0/
integer(4) rc
integer(4), external :: add
rc = add(%val(2), res, 3.14, 2.18,) ! The last argument is a
                                ! null argument.
if (rc == 0) then
print *, "res = ", res
else
print *, "number of arguments is invalid."
endif
end program
```

C program:

```
int add(int a, float *res, float *b, float *c, float *d)
{
    int ret = 0;
    if (a == 2)
        *res = *b + *c;
    else if (a == 3)
        *res = (*b + *c + *d);
    else
        ret = 1;
    return (ret);
}
```

sce | **nosce**

By default, the compiler performs short circuit evaluation in selected logical expressions using XL Fortran rules. Specifying **sce** allows the

compiler to use non-XL Fortran rules. The compiler will perform short circuit evaluation if the current rules allow it.

typestmt | **notypestmt**

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

typlessarg | **notyplessarg**

Converts all typeless constants to default integers if the constants are actual arguments to an intrinsic procedure whose associated dummy arguments are of integer type. Typeless actual arguments associated with dummy 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

- “-qintlog” on page 178
- “-qxlf77” on page 268
- See the section on the `INT` and `MOD` intrinsic functions in the *XL Fortran Language Reference* for further information.

-qposition

Category

Language element control

Purpose

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.

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`.

Syntax



@PROCESS:

```
@PROCESS POSITION({APPENDOLD | APPENDUNKNOWN} ...)
```

Defaults

The default setting depends on the I/O specifiers in the `OPEN` statement and on the compiler invocation command:

- **-qposition=appendold** for the `xlf`, `xlf_r`, and `f77/fort77` commands
- The defined Fortran 90 and Fortran 95 behaviors for the `xlf2003`, `xlf2003_r`, `xlf90`, `xlf90_r`, `xlf95`, `xlf95_r`, `f2003`, `f90`, and `f95` commands.

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

- *File positioning* in the *XL Fortran Optimization and Programming Guide*
- *OPEN statement* in the *XL Fortran Language Reference*

-qppsuborigarg

Category

Input control

@PROCESS

None.

Purpose

Instructs the C preprocessor to substitute original macro arguments before further macro expansion.

Syntax

►► -WF—, — -q noppsuborigarg
ppsuborigarg ◀◀

Defaults

- -qnoppsuborigarg

Usage

-qppsuborigarg is a C preprocessor option, and must therefore be specified using the **-WF** option.

Examples

Consider the following sample code, x.F:

```

#define PRINT_COMP(a) PRINT_4(SPLIT_COMP(a))
#define SPLIT_COMP(a) "Real:", real(a), "Imag:", imag(a)
#define PRINT_4(list) PRINT_LIST(list)
#define PRINT_LIST(list) print *, list

complex a
a = (3.5, -3.5)
PRINT_COMP(a)
end

```

If this code is compiled with **-qnoptionsuborigarg**, the C preprocessor reports an error because the parameter "list" in the function-like macro PRINT_4 is the expanded substitution text of the macro SPLIT_COMP(a). The C preprocessor therefore complains because PRINT_LIST is being called with four arguments but only expects one.

```

> xlf95 x.F -d
"x.F", line 8.1: 1506-215 (E) Too many arguments specified for macro PRINT_LIST.
** _main   === End of Compilation 1 ===
1501-510  Compilation successful for file x.F.
> cat Fx.f

```

```

complex a
a = (3.5, -3.5)
print *, "Real:"
end

```

When the code is compiled with **-qppsuborigarg**, the C preprocessor uses the text "SPLIT_COMP(a)" rather than the expanded substitution text of SPLIT_COMP(a) as the argument to the function-like macro PRINT_LIST. Only after the macro PRINT_LIST has been expanded, does the C preprocessor proceed to expand the macro "SPLIT_COMP(a)". As a result, the macro PRINT_LIST only receives the expected single argument "SPLIT_COMP(a)" rather than the four arguments.

```

> xlf95 x.F -d -WF,-qppsuborigarg
** _main   === End of Compilation 1 ===
1501-510  Compilation successful for file x.F.
> cat Fx.f

```

```

complex a
a = (3.5, -3.5)
print *, "Real:", real(a), "Imag:", imag(a)
end

```

Related information

- “-W” on page 284
- “-qfpp” on page 157
- “Passing Fortran files through the C preprocessor” on page 31

-qprefetch

Category

Optimization and tuning

@PROCESS

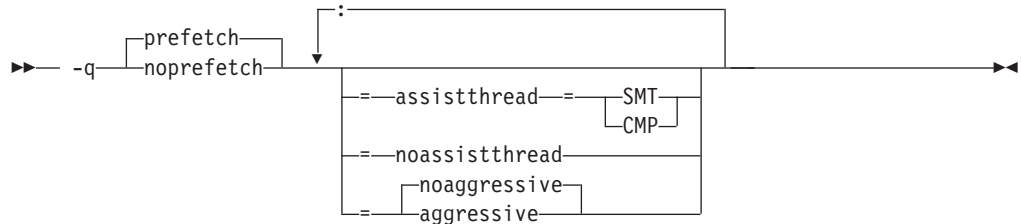
None.

Purpose

Inserts prefetch instructions automatically where there are opportunities to improve code performance.

When **-qprefetch** is in effect, the compiler may insert prefetch instructions in compiled code. When **-qnoprefetch** is in effect, prefetch instructions are not inserted in compiled code.

Syntax



Defaults

- **-qprefetch**
- **-qprefetch=noassistthread**
- **-qprefetch=noassistthread:noaggressive**

Parameters

assistthread | **noassistthread**

When you work with applications that generate a high cache-miss rate, you can use **-qprefetch=assistthread** to exploit assist threads for data prefetching. This suboption guides the compiler to exploit assist threads at optimization level **-O3 -qhot** or higher. If you do not specify **-qprefetch=assistthread**, **-qprefetch=noassistthread** is implied.

aggressive | **noaggressive**

This suboption guides the compiler to generate aggressive data prefetching at optimization level **-O3 -qhot** or higher. If you do not specify **aggressive**, **-qprefetch=noaggressive** is implied.

CMP

For systems based on the chip multi-processor architecture (CMP), you can use **-qprefetch=assistthread=cmp**.

SMT

For systems based on the simultaneous multi-threading architecture (SMT), you can use **-qprefetch=assistthread=smt**.

Note: If you do not specify either CMP or SMT, the compiler uses the default setting based on your system architecture.

Usage

The **-qnoprefetch** option does not prevent built-in functions such as **__prefetch_by_stream** from generating prefetch instructions.

When you run **-qprefetch=assistthread**, the compiler uses the delinquent load information to perform analysis and generates prefetching assist threads. The

delinquent load information can either be provided through the built-in `__mem_delay` function (const void *delinquent_load_address, const unsigned int delay_cycles), or gathered from dynamic profiling using `-qpdf1=level=2`.

When you use `-qpdf` to call `-qprefetch=assistthread`, you must use the traditional two-step PDF invocation:

1. Run `-qpdf1=level=2`
2. Run `-qpdf2 -qprefetch=assistthread`

Predefined macros

None.

Examples

```
DO I = 1, 1000
!IBM* MEM_DELAY(X(I), 10)
X(I) = X(I) + 1
END DO
```

Related information

- “-qarch” on page 120
- “-qhot” on page 167
- “-qpdf1, -qpdf2” on page 208
- “-qreport” on page 226
- MEM_DELAY section in the *XL Fortran Language Reference*

-qqcount

Category

Language element control

Purpose

Accepts the **Q** character-count edit descriptor (**Q**) as well as the extended-precision **Q** edit descriptor (**Qw.d**).

Syntax

►► — -q — noqcount
qcount —————►►

@PROCESS:

@PROCESS QCOUNT | NOQCOUNT

Defaults

With `-qnoqcount`, all **Q** edit descriptors are interpreted as the extended-precision **Q** edit descriptor.

Usage

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

- *Q (Character Count) Editing* in the *XL Fortran Language Reference*

-qrealsize

Category

Floating-point and integer control

Purpose

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 a useful alternative to the **-qautodbl** option in some situations.

Syntax

►► -qrealsize=4
8►►

@PROCESS:

@PROCESS REALSIZE(*bytes*)

Defaults

The default, **-qrealsize=4**, is suitable for programs that are written specifically for 32-bit computers.

Parameters

The allowed values for *bytes* are:

- 4
- 8

Usage

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.

-qautodbl is related to **-qrealsize**, although you cannot combine these options. When the **-qautodbl** option turns on automatic doubling, padding, or both, the **-qrealsize** option has no effect.

Setting **-qrealsize** to 8 overrides the setting of the **-qdpc** option.

In addition to type **REAL**, **-qrealsize** also works for type **vector(real)**.

Results

The option affects the sizes² of constants, variables, derived type components, and functions (which include intrinsic functions) for which no kind type parameter is specified. Objects that are declared with a kind type parameter or length, such as **REAL(4)** or **COMPLEX*16**, are not affected.

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.

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)
  VECTOR(REAL)         ! treated as a vector(real(8))
  VECTOR(REAL(4))      ! treated as a vector(real(4))
  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” on page 179 is a similar option that affects integer and logical objects.
- “-qautodbl” on page 126
- *Type declaration: type parameters and specifiers* in the *XL Fortran Language Reference*

2. In Fortran 90/95 terminology, these values are referred to as *kind type parameters*.

-qrecur

Category

Deprecated options

Purpose

Specifies whether external subprograms may be called recursively.

Not recommended.

Syntax

►► — -q — norecur
recur —►►

@PROCESS:

@PROCESS RECUR | NORECUR

Defaults

-qnorecur

Usage

For new programs, use the **RECURSIVE** keyword, which provides a standards-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. With the **RECURSIVE** keyword, you can specify exactly which procedures are recursive.

If you use the **xl**, **xl_r**, **f77**, or **fort77** command to compile programs that contain recursive calls, specify **-qnosave** to make the default storage class automatic.

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

```

-qreport

Category

Listings, messages, and compiler information

Purpose

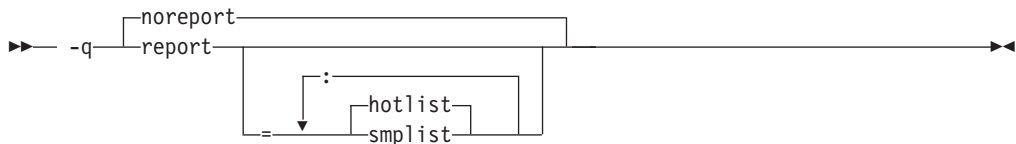
Produces listing files that show how sections of code have been optimized.

A listing file is generated with a .lst suffix for each source file named on the command line. When used with an option that enables automatic parallelization or vectorization, the listing file shows a pseudo-Fortran code listing and a summary of how program loops are parallelized or optimized. The report also includes diagnostic information to show why specific loops could not be parallelized or vectorized. For instance, when **-qreport** is used with **-qsimd=auto**, messages are provided to identify non-stride-one references that can prevent loop vectorization.

The compiler also reports the number of streams created for a given loop, which include both load and store streams. This information is included in the Loop Transformation section of the listing file. You can use this information to understand your application code and to tune your code for better performance. For example, you can distribute a loop which has more streams than the number supported by the underlying architecture. POWER4 and POWER5 support load stream prefetch and POWER6 supports both load and store stream prefetch.

Syntax

Option:



@PROCESS:

@PROCESS REPORT[({SMPLIST |HOTLIST}...)] | NOREPORT

Defaults

-qnoreport

Parameters

smplist | hotlist

When **-qreport=smplist** is in effect, produces a pseudo-Fortran listing that shows how the program is parallelized. This listing is produced before loop and other optimizations are performed. It includes messages that point out places in the program that can be modified to be more efficient. This report is only produced if **-qsmp** is in effect.

When **-qreport=hotlist** is in effect, produces a pseudo-Fortran listing that shows how loops are transformed, to assist you in tuning the performance of all loops. This report is only produced if **-qhot** is in effect.

In addition, if you specify the **-qreport=hotlist** option when **-qsmp** is in effect, a pseudo-Fortran listing will be produced that shows the calls to the SMP runtime library and the procedures created for parallel constructs.

Specifying **-qreport** with no suboptions is equivalent to **-qreport=hotlist**.

Usage

For **-qreport** to generate a loop transformation listing, you must also specify one of the following on the command line:

- **-qsimd=auto**
- **-qsmp**
- **-qhot=level=2** and **-qsmp**
- **-O5**
- **-qipa=level=2**

For **-qreport** to generate PDF information in the listing, you must specify the following option in the command line:

- **-qpdf2 -qreport**

For **-qreport** to generate a parallel transformation listing or parallel performance messages, you must also specify one of the following options on the command line:

- **-qsmp**
- **-O5**
- **-qipa=level=2**

To generate data reorganization information, specify the optimization level **-qipa=level=2** or **-O5** together with **-qreport**. Reorganizations include common block splitting, array splitting, array transposing, memory allocation merging, array interleaving, and array coalescing.

To generate information about data prefetch insertion locations, use the optimization level of **-qhot**, or any other option that implies **-qhot** together with **-qreport**. This information appears in the LOOP TRANSFORMATION SECTION of the listing file. In addition, when you use **-qprefetch=assistthread** to generate prefetching assist threads, the message: Assist thread for data prefetching was generated also appears in the LOOP TRANSFORMATION SECTION of the listing file.

To generate a list of aggressive loop transformations and parallelizations performed on loop nests in the LOOP TRANSFORMATION SECTION of the listing file, use the optimization level of **-qhot=level=2** and **-qsmp** together with **-qreport**.

The pseudo-Fortran code listing is not intended to be compilable. Do not include any of the pseudo-Fortran code in your program, and do not explicitly call any of the internal routines whose names may appear in the pseudo-Fortran code listing.

Examples

To compile `myprogram.f` so the compiler listing includes a report showing how loops are optimized, enter:

```
xlf -qhot -O3 -qreport myprogram.f
```

To compile `myprogram.c` so the compiler listing also includes a report showing how parallelized loops are transformed, enter:

```
xlf_r -qhot -qsmp -qreport=smp1ist myprogram.f
```

Related information

- “-qhot” on page 167
- “-qsimd” on page 234
- “-qipa” on page 181
- “-qsmp” on page 237
- “-qoptdebug” on page 206
- “-qprefetch” on page 220
- “Using -qoptdebug to help debug optimized programs” in the *XL Fortran Optimization and Programming Guide*

-qsaa

Category

Language element control

Purpose

Checks for conformance to the SAA FORTRAN language definition. It identifies nonconforming source code and also options that allow such nonconformances.

Syntax

►► -q nosaa
saa ◀◀

@PROCESS:

@PROCESS SAA | NOSAA

Defaults

-qnosaa

Usage

The **-qflag** option can override this option.

Use the **-qlanglvl** option to check your code for conformance to international standards.

Results

Warnings have a prefix of **(L)**, indicating a problem with the language level.

Related information

- “-qflag” on page 151
- “-qlanglvl” on page 187

-qsave

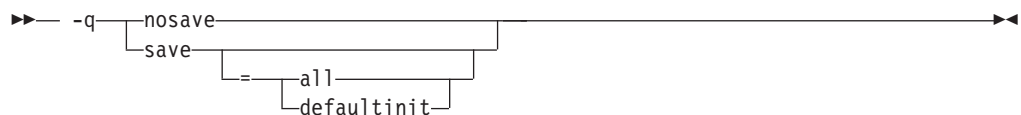
Category

Language element control

Purpose

Specifies the default storage class for local variables.

Syntax



@PROCESS:

```
@PROCESS SAVE[({ALL | DEFAULTINIT})] | NOSAVE
```

Defaults

When **-qnosave** is in effect, the default storage class is **AUTOMATIC**.

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 **xl f**, **xl f_r**, **f77**, and **fort77** commands have **-qsave** listed as a default option in `/opt/ibmcomp/xlf/13.1/etc/xlf.cfg` to preserve the previous behavior.

Parameters

The **-qsave** suboptions include:

all The default storage class is **STATIC**.

defaultinit

The default storage class is **STATIC** for variables of derived type that have default initialization specified, and **AUTOMATIC** otherwise.

The **all** and **defaultinit** suboptions are mutually exclusive.

Usage

The **-qnosave** option is usually necessary for multithreaded applications and subprograms that are compiled with the **-qrecur** option.

Examples

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

- “-qrecur” on page 225
- See *Storage classes for variables* in the *XL Fortran Language Reference* for information on how this option affects the storage class of variables.

-qsaveopt

Category

Object code control

@PROCESS

None.

Purpose

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

Syntax

```

▶▶ -q nosaveopt
saveopt

```

Defaults

-qnosaveopt

Usage

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

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

```

▶▶ @(#) opt { f | c | C } invocation options

```

```

▶▶ @(#) cfg config_file_options_list

```

```

▶▶ @(#) evn env_var_definition

```

where:

- f** Signifies a Fortran language compilation.
- c** Signifies a C language compilation.
- C** Signifies a C++ language compilation.

invocation

Shows the command used for the compilation, for example, `xlfc`.

options The list of command line options specified on the command line, with individual options separated by space.

config_file_options_list

The list of options specified by the **options** attribute in all configuration files that take effect in the compilation, separated by space.

env_var_definition

The environment variables that are used by the compiler. Currently only `XLFC_USR_CONFIG` is listed.

Note: You can always use this option, but the corresponding information is only generated when the environment variable `XLFC_USR_CONFIG` is set.

For more information about the environment variable `XLFC_USR_CONFIG`, see `XLFC_USR_CONFIG`.

Note: The string of the command-line options is truncated after 64k bytes.

Compiler version and release information, as well as the version and level of each component invoked during compilation, are also saved to the object file in the format:

```

▶▶ @(#) version { Version:--VV.RR.MMMM.LLLL | component_name--Version:--VV.RR--(product_name)--Level:--YYMMDD }

```

where:

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

component_name

Specifies the components that were invoked for this compilation, such as the low-level optimizer.

product_name

Indicates the product to which the component belongs (for example, C/C++ or Fortran).

YYMMDD

Represents the year, month, and date of the installed update. If the update installed is at the base level, the level is displayed as BASE.

If you want to simply output this information to standard output without writing it to the object file, use the **-qversion** option.

Examples

Compile `t.f` with the following command:

```
xlf t.f -c -qsaveopt -qhot
```

Issuing the **strings -a** command on the resulting `t.o` object file produces information similar to the following:

```
opt f /opt/ibmcmp/xlf/13.1/bin/xlf t.f -c -qsaveopt -qhot
cfg -qnozerosize -qsave -qalias=intptr -qposition=appendold
-qxlf90=noautodealloc:nosignedzero:oldpad
-qxlf77=intarg:intxor:persistent:noleadzero:gedit77:noblinkpad:oldboz:softeof
-qxlf2003=nopolymorphic:nobozlitargs:nostopexcept:novolatile:noautorealloc:oldnaninf -bh:4
version IBM XL Fortran for Linux, V13.1
version Version: 13.01.0000.0000
version Driver Version: 13.01(Fortran) Level: YYMMDD
version Fortran Front End and Run Time Version: 13.01(Fortran) Level: YYMMDD
version Fortran Transformer Version: 13.01(Fortran) Level: YYMMDD
version High-Level Optimizer Version: 11.01(C/C++) and 13.01(Fortran) Level: YYMMDD
version Low-Level Optimizer Version: 11.01(C/C++) and 13.01(Fortran) Level: YYMMDD
```

In the first line, `f` identifies the source used as Fortran, `/opt/ibmcmp/xlf/13.1/bin/xlf` shows the invocation command used, and `-qhot -qsaveopt` shows the compilation options. The second line, which starts with `cfg`, shows the compiler options added by the configuration file.

The remaining lines list each compiler component invoked during compilation, and its version and level. Components that are shared by multiple products may show more than one version number. Level numbers shown may change depending on the updates you have installed on your system.

Related information

- “-qversion” on page 264

-qsclk

Category

Language element control

@PROCESS

None.

Purpose

Specifies the resolution that the `SYSTEM_CLOCK` intrinsic procedure uses in a program.

Syntax

```
►► -q-sclk=centi  
          micro          ►►
```

Defaults

The default is centisecond resolution (`-qsclk=centi`). To use microsecond resolution, specify `-qsclk=micro`.

Related information

See `SYSTEM_CLOCK` in the *XL Fortran Language Reference* for more information on returning integer data from a real-time clock.

-qshowpdf

Category

Optimization and tuning

@PROCESS

None.

Purpose

When used with `-qpdf1` and a minimum optimization level of `-O2` at compile and link steps, inserts additional profiling information into the compiled application to collect call and block counts for all procedures in the application.

Syntax

```
►► -qnoshowpdf  
          showpdf          ►►
```

Defaults

`-qnoshowpdf`

Usage

When specified together with `-qpdf1`, the compiler inserts additional profiling information into the compiled application to collect call and block counts for all procedures in the application. Running the compiled application will record the call and block counts to the `._pdf` file.

After you run your application with training data, you can retrieve the contents of the `._pdf` file with the `showpdf` utility. This utility is described in “-qpdf1, -qpdf2” on page 208.

-qsigtrap

Category

Error checking and debugging

@PROCESS

None.

Purpose

Sets up the specified trap handler to catch **SIGTRAP** and **SIGFPE** exceptions when compiling a file that contains a main program.

This option enables you to install a handler for **SIGTRAP** or **SIGFPE** signals without calling the **SIGNAL** subprogram in the program.

Syntax

```
▶▶ -q-sigtrap [=-trap_handler-] ▶▶
```

Defaults

Not applicable.

Usage

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. To show more detailed information in the tracebacks generated by the trap handlers provided by XL Fortran (such as `xl__trce`), specify the **-qlinedebug** or **-g** option.

Related information

- “XL Fortran runtime exceptions” on page 47 describes the possible causes of exceptions.
- *Detecting and trapping floating-point exceptions in the XL Fortran Optimization and Programming Guide* describes a number of methods for dealing with exceptions that result from floating-point computations.
- *Installing an exception handler in the XL Fortran Optimization and Programming Guide* lists the exception handlers that XL Fortran supplies.

-qsimd

Category

Optimization and tuning

@PROCESS

None.

Purpose

Controls whether the compiler can automatically take advantage of vector instructions for processors that support them.

These instructions can offer higher performance when used with algorithmic-intensive tasks such as multimedia applications.

Syntax

►► -qsimd=noauto
auto ►►

Defaults

-qsimd=noauto

Usage

The **-qsimd=auto** option enables automatic generation of vector instructions for processors that support them. It replaces the **-qenablevmx** option, which has been deprecated.

When **-qsimd=auto** is in effect, the compiler converts certain operations that are performed in a loop on successive elements of an array into vector instructions. These instructions calculate several results at one time, which is faster than calculating each result sequentially. Applying this option is useful for applications with significant image processing demands.

The **-qsimd=noauto** option disables the conversion of loop array operations into vector instructions. Finer control can be achieved by using **-qstrict=ieefp**, **-qstrict=operationprecision**, and **-qstrict=vectorprecision**. For details, see “-qstrict” on page 247.

Note: Using vector instructions to calculate several results at one time might delay or even miss detection of floating point exceptions on some architectures. If detecting exceptions is important, do not use **-qsimd=auto**.

The following rules apply when you use the **-qsimd** option:

- Specifying the deprecated **-qenablevmx** option has the same effect as specifying **-qsimd=auto**. The compiler does not issue any warning for this.
- Specifying **-qsimd** without any suboption has the same effect as **-qsimd=auto**.
- This option is available only when you set **-qarch** to a target architecture that supports vector instructions.
- Specify **-qsimd=auto** only when your processor architecture supports vector processing.
- If you specify **-qsimd=auto** to enable IPA at the compile time but specify **-simd=noauto** at the link time, the compiler automatically sets **-qsimd=auto** and sets an appropriate value for **-qarch** to match the architecture specified at the compile time.

Related information

- “-qarch” on page 120
- “-qstrict” on page 247
- The NOSIMD directive in the *XL Fortran Language Reference*.

-qsmallstack

Category

Optimization and tuning

@PROCESS

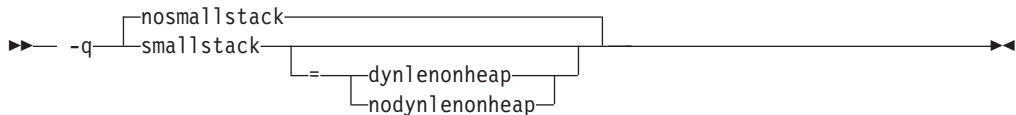
None.

Purpose

Minimizes stack usage where possible.

This compiler option controls two distinct, but related sets of transformations: general small stack transformations and dynamic length variable allocation transformations. These two kinds of transformations can be controlled independently of each other.

Syntax



Defaults

`-qnosmallstack`

Parameters

`dynlenonheap` | `nodynlenonheap`

The `-qsmallstack=dynlenonheap` suboption affects automatic objects that have nonconstant character lengths or a nonconstant array bound (DYNAMIC LENGTH ON HEAP). When specified, those automatic variables are allocated on the heap. When this suboption is not specified, those automatic variables are allocated on the stack.

Defaults

The default, `-qnosmallstack`, implies that all suboptions are off.

Usage

Using this option may adversely affect program performance; it should be used only for programs that allocate large amounts of data on the stack.

`-qsmallstack` with no suboptions enables only the general small stack transformations.

-qnosmallstack only disables the general small stack transformations. To disable **dynlenonheap** transformations, specify **-qsmallstack=nodynlenonheap** as well.

-qsmallstack=dynlenonheap enables the dynamic length variable allocation and general small stack transformations.

To enable only the **dynlenonheap** transformations, specify **-qsmallstack=dynlenonheap -qnosmallstack** .

When both **-qsmallstack** and **-qstacktemp** options are used, the **-qstacktemp** setting will be used to allocate applicable temporary variables if it is set to a non-zero value, even if this setting conflicts with that of **-qsmallstack**. The **-qsmallstack** setting will continue to apply transformations not affected by **-qstacktemp**.

Related information

- “-qstacktemp” on page 245

-qsmp

Category

Optimization and tuning

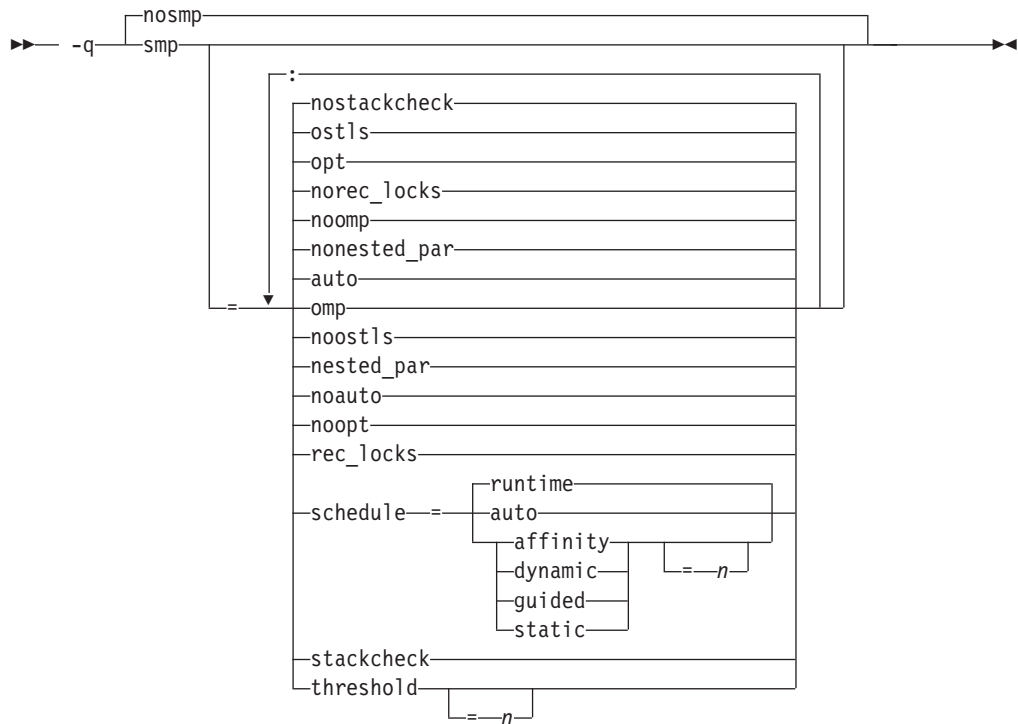
@PROCESS

None.

Purpose

Enables parallelization of program code.

Syntax



Defaults

-qnosmp. Code is produced for a uniprocessor machine.

Parameters

auto | noauto

Enables or disables automatic parallelization and optimization of program code. By default, the compiler will attempt to parallelize explicitly coded DO loops as well as those that are generated by the compiler for array language. When **noauto** is in effect, only program code explicitly parallelized with OpenMP directives is optimized. **noauto** is implied if you specify **-qsmp=omp** or **-qsmp=noopt**.

nested_par | nonested_par

By default, the compiler serializes a nested parallel construct. When **nested_par** is in effect, the compiler parallelizes prescriptive nested parallel constructs (PARALLEL DO, PARALLEL SECTIONS). This includes not only the loop constructs that are nested within a scoping unit but also parallel constructs in subprograms that are referenced (directly or indirectly) from within other parallel constructs. Note that this suboption has no effect on loops that are automatically parallelized. In this case, at most one loop in a loop nest (in a scoping unit) will be parallelized. **nested_par** does not provide true nested parallelism because it does not cause new team of threads to be created for nested parallel regions. Instead, threads that are currently available are reused.

This suboption should be used with caution. Depending on the number of threads available and the amount of work in an outer loop, inner loops could be executed sequentially even if this option is in effect. Parallelization overhead may not necessarily be offset by program performance gains.

Note:

- The implementation of the **nested_par** suboption does not comply with the OpenMP API. There is no support for OpenMP nested parallelism. As such, the **omp_get_nested** routine always returns false according to the OpenMP API.
- If you specify this suboption, the runtime library uses the same threads for the nested PARALLEL DO and PARALLEL SECTIONS constructs that it used for the enclosing PARALLEL constructs.

omp | noomp

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

Specifying **omp** also has the following effects:

- Automatic parallelization is disabled.
- All previously recognized directive triggers are ignored. The only recognized directive trigger is \$OMP. However, you can specify additional triggers on subsequent **-qdirective** options.
- The **-qclines** compiler option is enabled.
- When the C preprocessor is invoked, the `_OPENMP` C preprocessor macro is defined automatically, with the value 200505, which is useful in supporting conditional compilation. See *Conditional Compilation* in the *XL Fortran Language Reference* for more information.

opt | noopt

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

ostls | noostls

Enables Thread Local Storage (TLS) provided by the operating system to be used for **threadprivate** data. You can use the **noostls** suboption to enable the non-TLS for **threadprivate**. The **noostls** suboption is provided for backward compatibility.

Note: If you want to use this suboption, your operating system must support TLS to implement OpenMP **threadprivate** data. Use **noostls** to disable OS level TLS if your operating system does not support it.

rec_locks | norec_locks

Determines whether recursive locks are used to avoid problems associated with CRITICAL constructs. When **rec_locks** is in effect, nested critical sections will not cause a deadlock; a thread can enter a CRITICAL construct from within the dynamic extent of another CRITICAL construct that has the same name. Note that the **rec_locks** suboption specifies behavior for critical constructs that is inconsistent with the OpenMP API.

schedule

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

affinity[=*n*]

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

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

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

The **affinity** scheduling type does not appear in the OpenMP API standard.

auto

Scheduling of the loop iterations is delegated to the compiler and runtime systems. The compiler and runtime system can choose any possible mapping of iterations to threads (including all possible valid schedule types) and these might be different in different loops. Do not specify chunk size (*n*).

dynamic[=*n*]

The iterations of a loop are divided into chunks containing *n* iterations each. If *n* is not specified, then the chunks consist of **ceiling**(*number_of_iterations*/*number_of_threads*). iterations.

Active threads are assigned these chunks on a "first-come, first-do" basis. Chunks of the remaining work are assigned to available threads until all work has been assigned.

If a thread is asleep, its assigned work will be taken over by an active thread once that thread becomes available.

guided[=*n*]

The iterations of a loop are divided into progressively smaller chunks until a minimum chunk size of *n* loop iterations is reached. If *n* is not specified, the default value for *n* is 1 iteration.

Active threads are assigned chunks on a "first-come, first-do" basis. The first chunk contains **ceiling**(*number_of_iterations*/*number_of_threads*) iterations. Subsequent chunks consist of **ceiling**(*number_of_iterations_left* / *number_of_threads*) iterations.

runtime

Specifies that the chunking algorithm will be determined at run time.

static[=*n*]

The iterations of a loop are divided into chunks containing *n* iterations each. Each thread is assigned chunks in a "round-robin" fashion. This is known as *block cyclic scheduling*. If the value of *n* is 1, then the scheduling type is specifically referred to as *cyclic scheduling*.

If *n* is not specified, the chunks will contain **ceiling**(*number_of_iterations*/*number_of_threads*) iterations. Each thread is assigned one of these chunks. This is known as *block scheduling*.

If a thread is asleep and it has been assigned work, it will be awakened so that it may complete its work.

n Must be an integer of value 1 or greater.

Specifying **schedule** with no suboption is equivalent to **schedule=runtime**.

For more information on chunking algorithms and SCHEDULE, refer to *Directives* in the *XL Fortran Language Reference*.

stackcheck | **nostackcheck**

Causes the compiler to check for stack overflow by slave threads at run time, and issue a warning if the remaining stack size is less than the number of bytes specified by the **stackcheck** option of the XLSMPOPTS environment variable. This suboption is intended for debugging purposes, and only takes effect when **XLSMPOPTS=stackcheck** is also set; see XLSMPOPTS in the *XL Fortran Optimization and Programming Guide* for more information.

threshold[=*n*]

When **-qsmp=auto** is in effect, controls the amount of automatic loop parallelization that occurs. The value of *n* represents the minimum amount of work required in a loop in order for it to be parallelized. Currently, the calculation of "work" is weighted heavily by the number of iterations in the loop. In general, the higher the value specified for *n*, the fewer loops are parallelized. Specifying a value of 0 instructs the compiler to parallelize all auto-parallelizable loops, whether or not it is profitable to do so. Specifying a value of 100 instructs the compiler to parallelize only those auto-parallelizable loops that it deems profitable. Specifying a value of greater than 100 will result in more loops being serialized.

n Must be a positive integer of 0 or greater.

If you specify **threshold** with no suboption, the program uses a default value of 100.

Specifying **-qsmp** without suboptions is equivalent to:

```
-qsmp=auto:opt:noomp:norec_locks:nonested_par:schedule=runtime:  
nostackcheck:threshold=100:ostls
```

Usage

- Specifying the **omp** suboption always implies **noauto**. Specify **-qsmp=omp:auto** to apply automatic parallelization on OpenMP-compliant applications, as well.
- When **-qsmp** is in effect, the compiler recognizes all directives with the trigger constants SMP\$, \$OMP, and IBMP, unless you specify the **omp** suboption. If you specify **omp** and want the compiler to recognize directives specified with the other triggers, you can use the **-qdirective** option to do so.
- You should only use **-qsmp** with the **_r**-suffixed invocation commands, to automatically link in all of the threadsafe components. You can use the **-qsmp** option with the non-**_r**-suffixed invocation commands, but you are responsible for linking in the appropriate components. . If you use the **-qsmp** option to compile any source file in a program, then you must specify the **-qsmp** option at link time as well, unless you link by using the **ld** command.
- If you use the **f77** or **fort77** command with the **-qsmp** option to compile programs, specify **-qnosave** to make the default storage class automatic, and specify **-qthreaded** to tell the compiler to generate threadsafe code.
- Object files generated with the **-qsmp=opt** option can be linked with object files generated with **-qsmp=noopt**. The visibility within the debugger of the variables in each object file will not be affected by linking.

- Specifying **-qsmp** implicitly sets **-O2**. The **-qsmp** option overrides **-qnooptimize**, but does not override **-O3**, **-O4**, or **-O5**. When debugging parallelized program code, you can disable optimization in parallelized program code by specifying **qsmp=noopt**.
- The **-qsmp=noopt** suboption overrides performance optimization options anywhere on the command line unless **-qsmp** appears after **-qsmp=noopt**. For example, **-qsmp=noopt -O3** is equivalent to **-qsmp=noopt**, while **-qsmp=noopt -O3 -qsmp** is equivalent to **-qsmp -O3**.

Examples

In the following example, you should specify **-qsmp=rec_locks** to avoid a deadlock caused by critical constructs.

```

program t
  integer i, a, b

  a = 0
  b = 0
!smp$ parallel do
  do i=1, 10
!smp$ critical
  a = a + 1
!smp$ critical
  b = b + 1
!smp$ end critical
!smp$ end critical
  enddo
end

```

Related information

- “-O” on page 108
- “-qthreaded” on page 258
- XLSMPOPTS environment variable and Parallelization directives in the *XL Fortran Optimization and Programming Guide*

-qsource

Category

Listings, messages, and compiler information

Purpose

Produces a compiler listing file that includes the source section of the listing and provides additional source information when printing error messages.

Syntax

►► -q nosource
source ◀◀

@PROCESS:

@PROCESS SOURCE | NOSOURCE

Defaults

-qnosource

Usage

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

- “Listings, messages, and compiler information” on page 85
- “Source section” on page 302

-qspillsize

Category

Compiler customization

Purpose

-qspillsize is the long form of **-NS**. See “-NS” on page 107.

Syntax

► -q spillsize=*bytes* ◀

@PROCESS:

@PROCESS SPILLSIZE(*bytes*)

Defaults

Not applicable.

-qstackprotect

Category

“Object code control” on page 82

@PROCESS

None.

Purpose

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

Syntax

► -q ┌ nostackprotect
└ stackprotect = ┌ all
└ size=*N* ◀

Defaults

- -qnostackprotect

Parameters

all

all protects all procedures whether or not there are vulnerable objects. This option is not set by default.

size=*N*

size=*N* protects all procedures containing automatic objects greater or equal to *N* bytes in size. The default size is 8 when **-qstackprotect** is enabled.

Note: When both **all** and **size** are used, the last option wins.

Usage

-qstackprotect generates extra code to protect procedures with vulnerable objects against stack corruption. This option is disabled by default because it can cause performance degradation. The default option is **-qnostackprotect**.

To generate code to protect all procedures with vulnerable objects:

```
xlf myprogram.f -qstackprotect=all
```

To generate code to protect procedures with objects of certain bytes:

```
xlf myprogram.f -qstackprotect=size=8
```

Note:

- This option cannot be used with @PROCESS options.
- Because of the dependency on **glibc** in Linux, this option requires the following Linux levels:
 - Linux OS with **GLIBC** 2.4 and up (Machines with **GCC** 4.X and up).
- All supported Linux systems already support this feature.

-qstacktemp

Category

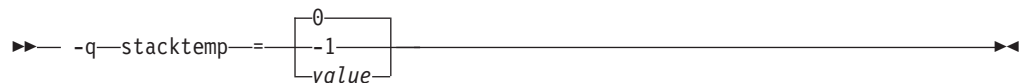
Optimization and tuning

Purpose

Determines where to allocate certain XL Fortran compiler temporaries at run time.

Applicable compiler temporaries are the set of temporary variables created by the compiler for its own use when it determines it can safely apply these. Most typically, the compiler creates these kinds of temporaries to hold copies of XL Fortran arrays for array language semantics, especially in conjunction with calls to intrinsic functions or user subprograms.

Syntax



@PROCESS:

```
@PROCESS STACKTEMP={0 | -1 | positive integer value}
```

Defaults

-qstacktemp=0

Parameters

The possible suboptions are:

- 0** Based on the target environment, the compiler determines whether it will allocate applicable temporaries on the heap or the stack. If this setting causes your program to run out of stack storage, try specifying a nonzero value instead, or try using the **-qsmallstack** option.
- 1** Allocates applicable temporaries on the stack. Generally, this is the best performing setting but uses the most amount of stack storage.
- value** Allocates applicable temporaries less than *value* on the stack and those greater than or equal to *value* on the heap. *value* is a positive integer. A value of 1 Mb has been shown to be a good compromise between stack

storage and performance for many programs, but you may need to adjust this number based on your application's characteristics.

Usage

If you have programs that make use of large arrays, you may need to use this option to help prevent stack space overflow when running them. For example, for SMP or OpenMP applications that are constrained by stack space, you can use this option to move some compiler temporaries onto the heap from the stack.

The compiler cannot detect whether or not the stack limits will be exceeded when an application runs. You will need to experiment with several settings before finding the one that works for your application. To override an existing setting, you must specify a new setting.

The `-qstacktemp` option can take precedence over the `-qsmallstack` option for certain compiler-generated temporaries.

Related information

- “`-qsmallstack`” on page 236

-qstaticlink

Category

Linking

@PROCESS directive

None.

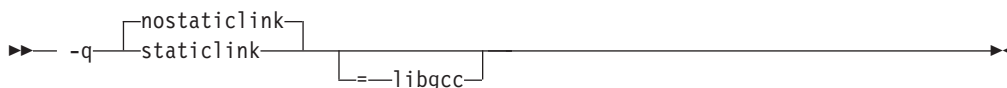
Purpose

Controls how shared and nonshared runtime libraries are linked into an application.

When `-qstaticlink` is in effect, the compiler links only static libraries with the object file being produced. When `-qnostaticlink` is in effect, the compiler links shared libraries with the object file being produced.

This option provides the ability to specify linking rules that are equivalent to those implied by the GNU options `-static`, `-static-libgcc`, and `-shared-libgcc`, used singly and in combination.

Syntax



Defaults

`-qnostaticlink`

Parameters

libgcc

When **libgcc** is specified together with **nostaticlink**, the compiler links to the shared version of **libgcc**. When **libgcc** is specified together with **staticlink**, the compiler links to the static version of **libgcc**.

Usage

Important: Any use of third-party libraries or products is subject to the provisions in their respective licenses. Using the **-qstaticlink** option can have significant legal consequences for the programs that you compile. It is strongly recommended that you seek legal advice before using this option.

The following table shows the equivalent GNU and XL Fortran options for specifying linkage of shared and nonshared libraries.

Table 20. Option mappings: control of the GNU linker

GNU option	Meaning	XL Fortran option
-shared	Build a shared object.	-qmkshrobj
-static	Build a static object and prevent linking with shared libraries. Every library linked to must be a static library.	-qstaticlink
-shared-libgcc	Link with the shared version of libgcc.	-qnostaticlink or -qnostaticlink=libgcc (these two are identical) Note: This is the default setting on SUSE Linux Enterprise Server (SLES) and Red Hat Enterprise Linux (RHEL).
-static-libgcc	Link with the static version of libgcc. You can still link your shared libraries.	-qstaticlink=libgcc

Note: Options **-qmkshrobj** and **-qstaticlink** are incompatible and cannot be specified together.

Related information

- “-qmkshrobj” on page 200

-qstrict

Category

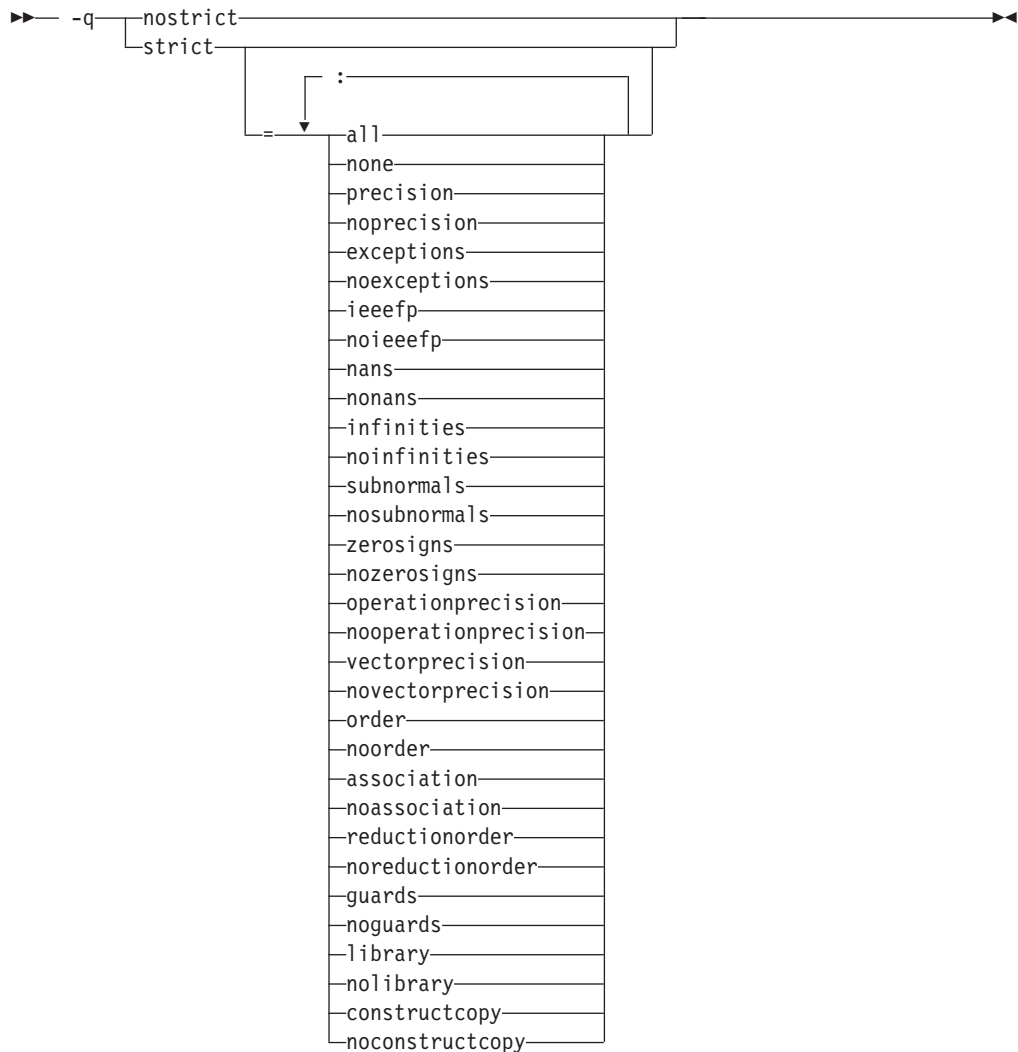
Optimization and tuning

Purpose

Ensures that optimizations done by default at optimization levels **-O3** and higher, and, optionally at **-O2**, do not alter certain program semantics mostly related to strict IEEE floating-point conformance.

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

Syntax



@PROCESS:

@PROCESS STRICT[(suboptions)] | NOSTRICT

Defaults

- Always **-qstrict** or **-qstrict=all** when the **-qnoot** or **-O0** optimization level is in effect
- **-qstrict** or **-qstrict=all** is the default when the **-O2** or **-O** optimization level is in effect
- **-qnostrict** or **-qstrict=none** is the default when **-O3** or a higher optimization level is in effect

Parameters

The **-qstrict** suboptions include the following:

all | none

all disables all semantics-changing transformations, including those controlled by the **ieeefp**, **order**, **library**, **constructcopy**, **precision**, and **exceptions** suboptions. **none** enables these transformations.

precision | noprecision

precision disables all transformations that are likely to affect floating-point precision, including those controlled by the **subnormals**, **operationprecision**, **vectorprecision**, **association**, **reductionorder**, and **library** suboptions. **noprecision** enables these transformations.

exceptions | noexceptions

exceptions disables all transformations likely to affect exceptions or be affected by them, including those controlled by the **nans**, **infinities**, **subnormals**, **guards**, **library**, and **constructcopy** suboptions. **noexceptions** enables these transformations.

ieeeFP | noieeeFP

ieeeFP disables transformations that affect IEEE floating-point compliance, including those controlled by the **nans**, **infinities**, **subnormals**, **zerosigns**, **vectorprecision**, and **operationprecision** suboptions. **noieeeFP** enables these transformations.

nans | nonans

nans disables transformations that may produce incorrect results in the presence of, or that may incorrectly produce IEEE floating-point NaN (not-a-number) values. **nonans** enables these transformations.

infinities | noinfinities

infinities disables transformations that may produce incorrect results in the presence of, or that may incorrectly produce floating-point infinities. **noinfinities** enables these transformations.

subnormals | nosubnormals

subnormals disables transformations that may produce incorrect results in the presence of, or that may incorrectly produce IEEE floating-point subnormals (formerly known as denorms). **nosubnormals** enables these transformations.

zerosigns | nozerosigns

zerosigns disables transformations that may affect or be affected by whether the sign of a floating-point zero is correct. **nozerosigns** enables these transformations.

operationprecision | nooperationprecision

operationprecision disables transformations that produce approximate results for individual floating-point operations. **nooperationprecision** enables these transformations.

vectorprecision | novectorprecision

vectorprecision disables vectorization in loops where it might produce different results in vectorized iterations than in nonvectorized residue iterations. **vectorprecision** ensures that every loop iteration of identical floating point operations on identical data produces identical results.

novectorprecision enables vectorization even when different iterations might produce different results from the same inputs.

order | noorder

order disables all code reordering between multiple operations that may affect results or exceptions, including those controlled by the **association**, **reductionorder**, and **guards** suboptions. **noorder** enables code reordering.

association | noassociation

association disables reordering operations within an expression. **noassociation** enables reordering operations.

reductionorder | **noreductionorder**

reductionorder disables parallelizing floating-point reductions.
noreductionorder enables parallelizing these reductions.

guards | **noguards**

guards disables moving operations past guards (that is, past **IF** statements, out of loops, or past subroutine or function calls that might end the program) which control whether the operation should be executed. **noguards** enables moving operations past guards.

library | **nolibrary**

library disables transformations that affect floating-point library functions; for example, transformations that replace floating-point library functions with other library functions or with constants. **nolibrary** enables these transformations.

constructcopy | **noconstructcopy**

constructcopy disables constructing arrays in place instead of using a temporary copy where an exception could occur. **noconstructcopy** enables constructing such arrays.

Usage

The **all**, **precision**, **exceptions**, **ieeefp**, and **order** suboptions and their negative forms are group suboptions that affect multiple, individual suboptions. For many situations, the group suboptions will give sufficient granular control over transformations. Group suboptions act as if either the positive or the no form of every suboption of the group is specified. Where necessary, individual suboptions within a group (like **subnormals** or **operationprecision** within the **precision** group) provide control of specific transformations within that group.

With **-qnostrict** or **-qstrict=none** in effect, the following optimizations are turned on:

- Code that may cause an exception may be rearranged. The corresponding exception might happen at a different point in execution or might not occur at all. (The compiler still tries to minimize such situations.)
- Floating-point operations may not preserve the sign of a zero value. (To make certain that this sign is preserved, you also need to specify **-qfloat=rrm**, **-qfloat=nomaf**, or **-qfloat=strictnmaf**.)
- Floating-point expressions may be reassociated. For example, $(2.0*3.1)*4.2$ might become $2.0*(3.1*4.2)$ if that is faster, even though the result might not be identical.
- The **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.

Specifying various **-qstrict[=suboptions]** or **-qnostrict** combinations sets the following suboptions:

- **-qstrict** or **-qstrict=all** sets **-qfloat=norsqrt:rngchk**. **-qnostrict** or **-qstrict=none** sets **-qfloat=rsqrt:norngchk**.
- **-qstrict=operationprecision** or **-qstrict=exceptions** sets **-qfloat=nofltint**. Specifying both **-qstrict=nooperationprecision** and **-qstrict=noexceptions** sets **-qfloat=fltint**.
- **-qstrict=infinities**, **-qstrict=operationprecision**, or **-qstrict=exceptions** sets **-qfloat=norsqrt**.

- `-qstrict=noinfinities:nooprecision:noexceptions` sets `-qfloat=rsqrt`.
- `-qstrict=nans`, `-qstrict=infinities`, `-qstrict=zerosigns`, or `-qstrict=exceptions` sets `-qfloat=rngchk`. Specifying all of `-qstrict=nonans:nozerosigns:noexceptions` or `-qstrict=noinfinities:nozerosigns:noexceptions`, or any group suboptions that imply all of them, sets `-qfloat=norngchk`.

Note: For details about the relationship between `-qstrict` suboptions and their `-qfloat` counterparts, see “`-qfloat`” on page 152.

To override any of these settings, specify the appropriate `-qfloat` suboptions after the `-qstrict` option on the command line.

Examples

To compile `myprogram.f` so that the aggressive optimizations of `-O3` are turned off, and division by the result of a square root is replaced by multiplying by the reciprocal (`-qfloat=rsqrt`), enter:

```
xlf myprogram.f -O3 -qstrict -qfloat=rsqrt
```

To enable all transformations except those affecting precision, specify:

```
xlf myprogram.f -qstrict=none:precision
```

To disable all transformations except those involving NaNs and infinities, specify:

```
xlf myprogram.f -qstrict=all:nonans:noinfinities
```

Related information

- “`-qsimd`” on page 234
- “`-qessl`” on page 146
- “`-qfloat`” on page 152
- “`-qhot`” on page 167
- “`-O`” on page 108
- “`-qxf90`” on page 270

`-qstrictieemod`

Category

Floating-point and integer control

Purpose

Specifies whether the compiler will adhere to the Fortran 2003 IEEE arithmetic rules for the `ieee_arithmetic` and `ieee_exceptions` intrinsic modules.

Syntax

```

>> -q strictieemod | nostrictieemod <<<

```

@PROCESS:

@PROCESS STRICTIEEMOD | NOSTRICTIEEMOD

Defaults

-qstrictieemod

Usage

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

Category

Optimization and tuning

@PROCESS

None.

Purpose

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

Syntax

►► -q nostrict_induction
strict_induction ◀◀

Defaults

-qnostrict_induction

Usage

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.

Related information

- “-O” on page 108

-qsuffix

Category

Input control

@PROCESS

None.

Purpose

Specifies the source-file suffix on the command line.

This option saves time for the user by permitting files to be used as named with minimal makefile modifications. Only one setting is supported at any one time for any particular file type.

Syntax

```
►► -q-suffix=                    f=          source-file-suffix          ►►
                          o=          object-file-suffix          ►►
                          s=          assembler-source-file-suffix  ►►
                          cpp=          preprocessor-source-file-suffix►►
```

Defaults

Not applicable.

Parameters

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.

Examples

For instance,

```
xlf a1.f2k a2.F2K -qsuffix=f=f2k:cpp=F2K
```

will cause these effects:

- The compiler is invoked for source files with a suffix of .f2k and .F2K.
- cpp is invoked for files with a suffix of .F2K.

-qsuppress

Category

Listings, messages, and compiler information

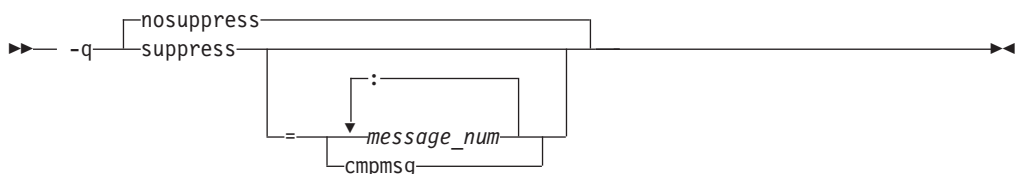
@PROCESS

None.

Purpose

Prevents specific informational or warning messages from being displayed or added to the listing file, if one is generated.

Syntax



Defaults

Not applicable.

Parameters

message_num[:message_num ...]

Suppresses the display of a specific compiler message (*nnnn-mmm*) or a list of messages (*nnnn-mmm[:nnnn-mmm ...]*). To suppress a list of messages, separate each message number with a colon.

nnnn-mmm is the message number, where:

- *nnnn* must be a four-digit integer between 1500 and 1585; this is the range of XL Fortran message numbers.
- *mmm* must be any three-digit integer (with leading zeros if necessary).

cmpmsg

Suppresses the informational messages that report compilation progress and a successful completion.

This suboption has no effect on any error messages that are emitted.

Usage

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.

Note:

- 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 with **-qsuppress**.
- To suppress IPA messages, enter **-qsuppress** before **-qipa** on the command line.

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” on page 151.

-qswapomp

Category

Portability and migration

Purpose

Specifies that the compiler should recognize and substitute OpenMP routines in XL Fortran programs.

The OpenMP routines for Fortran and C have different interfaces. To support multi-language applications that use OpenMP routines, the compiler needs to recognize OpenMP routine names and substitute them with the XL Fortran versions of these routines, regardless of the existence of other implementations of such routines.

Syntax

```
► — -q — ◄
```

@PROCESS:

@PROCESS SWAPOMP | NOSWAPOMP

Defaults

-qswapomp

Usage

The compiler does not perform substitution of OpenMP routines when you specify the **-qnoswapomp** option.

The **-qswapomp** and **-qnoswapomp** options only affect Fortran subprograms that reference OpenMP routines that exist in the program.

Rules

- If a call to an OpenMP routine resolves to a dummy procedure, module procedure, an internal procedure, a direct invocation of a procedure itself, or a statement function, the compiler will not perform the substitution.
- When you specify an OpenMP routine, the compiler substitutes the call to a different special routine depending upon the setting of the **-qintsize** option. In this manner, OpenMP routines are treated as generic intrinsic procedures.
- Unlike generic intrinsic procedures, if you specify an OpenMP routine in an **EXTERNAL** statement, the compiler will not treat the name as a user-defined external procedure. Instead, the compiler will still substitute the call to a special routine depending upon the setting of the **-qintsize** option.
- An OpenMP routine cannot be extended or redefined, unlike generic intrinsic procedures.

Examples

In the following example, the OpenMP routines are declared in an **INTERFACE** statement.

```
@PROCESS SWAPOMP

      INTERFACE
      FUNCTION OMP_GET_THREAD_NUM()
        INTEGER OMP_GET_THREAD_NUM
      END FUNCTION OMP_GET_THREAD_NUM

      FUNCTION OMP_GET_NUM_THREADS()
        INTEGER OMP_GET_NUM_THREADS
      END FUNCTION OMP_GET_NUM_THREADS
      END INTERFACE

      IAM = OMP_GET_THREAD_NUM()
      NP = OMP_GET_NUM_THREADS()
      PRINT *, IAM, NP
      END
```

Related information

See the *OpenMP execution environment, lock and timing routines* section in the *XL Fortran Optimization and Programming Guide*.

-qtbtable

Category

Object code control

@PROCESS

None.

Purpose

Controls the amount of debugging traceback information that is included in the object files.

Note: Applies to the 64-bit environment only.

Syntax

►► -q-tbtable =

full
none
small

 ◀◀

Defaults

Not applicable.

Parameters

full The object code contains full traceback information. The program is debuggable, and if it stops because of a runtime exception, it produces a traceback listing that includes the names of all procedures in the call chain.

none The object code contains no traceback information at all. You cannot debug the program, because a debugger or other code-examination tool cannot unwind the program's stack at run time. If the program stops because of a runtime exception, it does not explain where the exception occurred.

small The object code contains traceback information but not the names of procedures or information about procedure parameters. You can debug the program, but some non-essential information is unavailable to the debugger. If the program stops because of a runtime exception, it explains where the exception occurred but reports machine addresses rather than procedure names.

Defaults

- Code compiled with **-g** or without **-O** has full traceback information (**-qtbtable=full**).
- Code compiled with **-O** or higher optimization contains less traceback information (**-qtbtable=small**).

Usage

This option is most suitable for programs that contain many long procedure names, such as the internal names constructed for module procedures. You may find it more applicable to C++ programs than to Fortran programs.

You can use this option to make your program smaller, at the cost of making it harder to debug. When you reach the production stage and want to produce a program that is as compact as possible, you can specify **-qtbtable=none**. Otherwise, the usual defaults apply.

Related information

- “-g” on page 103
- “-qcompact” on page 134
- “-O” on page 108
- *Debugging optimized code* in the *XL Fortran Optimization and Programming Guide*

-qthreaded

Category

Object code control

@PROCESS

None.

Purpose

Indicates to the compiler whether it must generate threadsafe code.

Syntax

▶▶ — -q—threaded —————▶▶

Defaults

-qthreaded is the default for the **xlf_r**, **xlf90_r**, **xlf95_r**, and **xlf2003_r** commands.

Usage

Specifying the **-qthreaded** option implies **-qdirective=ibmt**, and by default, the *trigger_constant* **IBMT** is recognized.

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 threadsafe code. Specifying these options ensures that variables and code created by the compiler are threadsafe; it does not guarantee the thread safety of user-written code.

If you use the **ENTRY** statement to have an alternate entry point for a subprogram and the **xlf_r** command to compile, also specify the **-qxlf77=nopersistent** option to be threadsafe. You should implement the appropriate locking mechanisms, as well.

-qtimestamps

Category

“Output control” on page 77

@PROCESS

None.

Purpose

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

Syntax

► — -q timestamps
notimestamps —►

Defaults

-qtimestamps

Usage

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

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

-qtune

Category

Optimization and tuning

@PROCESS

None.

Purpose

Tunes instruction selection, scheduling, and other architecture-dependent performance enhancements to run best on a specific hardware architecture.

Syntax



Defaults

-qtune=balanced when the default **-qarch** setting is in effect. Otherwise, the default depends on the effective **-qarch** setting. See Table 21 on page 261 for details.

Parameters

auto

Optimizations are tuned for the platform on which the application is compiled.

balanced

Optimizations are tuned across a selected range of recent hardware.

ppc970

Optimizations are tuned for the PowerPC 970 processor.

pwr3

Optimizations are tuned for the POWER3 hardware platforms.

pwr4

Optimizations are tuned for the POWER4 hardware platforms.

pwr5

Optimizations are tuned for the POWER5 hardware platforms.

pwr6

Optimizations are tuned for the POWER6 hardware platforms.

pwr7

Optimizations are tuned for the POWER7 hardware platforms.

rs64b

Optimizations are tuned for the RS64II processor.

rs64c

Optimizations are tuned for the RS64III processor.

Usage

If you want your program to run on more than one architecture, but to be tuned to a particular architecture, you can use a combination of the **-qarch** and **-qtune** options. These options are primarily of benefit for floating-point intensive programs.

By arranging (scheduling) the generated machine instructions to take maximum advantage of hardware features such as cache size and pipelining, **-qtune** can improve performance. It only has an effect when used in combination with options that enable optimization.

Although changing the **-qtune** setting may affect the performance of the resulting executable, it has no effect on whether the executable can be executed correctly on a particular hardware platform.

Acceptable combinations of **-qarch** and **-qtune** are shown in the following table.

Table 21. Acceptable -qarch/-qtune combinations

-qarch option	Default -qtune setting	Available -qtune settings
ppc	balanced	auto rs64b rs64c pwr3 pwr4 pwr5 pwr6 pwr7 ppc970 balanced
ppcgr	balanced	auto rs64b rs64c pwr3 pwr4 pwr5 pwr6 pwr7 ppc970 balanced
ppc64	balanced	auto rs64b rs64c pwr3 pwr4 pwr5 pwr6 pwr7 ppc970 balanced
ppc64gr	balanced	auto rs64b rs64c pwr3 pwr4 pwr5 pwr6 pwr7 ppc970 balanced
ppc64grsq	balanced	auto rs64b rs64c pwr3 pwr4 pwr5 pwr6 pwr7 ppc970 balanced
ppc64v	ppc970	auto ppc970 pwr6 balanced
ppc970	ppc970	auto ppc970 balanced
pwr3	pwr3	auto pwr3 pwr4 pwr5 pwr7 ppc970 balanced
pwr4	pwr4	auto pwr4 pwr5 pwr7 ppc970 balanced
pwr5	pwr5	auto pwr5 pwr7 balanced
pwr5x	pwr5	auto pwr5 pwr7 balanced
pwr6	pwr6	auto pwr6 pwr7 balanced
pwr6e	pwr6	auto pwr6 balanced
pwr7	pwr7	auto pwr7 balanced
rs64b	rs64b	auto rs64b
rs64c	rs64c	auto rs64c

Examples

To specify that the executable program testing compiled from myprogram.f is to be optimized for a POWER7 hardware platform, enter:

```
xlf -o testing myprogram.f -qtune=pwr7
```

Related information

- “-qarch” on page 120
- "Optimizing your applications" in the *XL Fortran Optimization and Programming Guide*

-qundef

Category

Language element control

Purpose

-qundef is the long form of the “-u” on page 282 option.

Syntax

►► -q noundef undef ◀◀

@PROCESS:

@PROCESS UNDEF | NOUNDEF

Defaults

-qnoundef

-qunroll

Category

Optimization and tuning

@PROCESS

None.

Purpose

Specifies whether unrolling **DO** loops is allowed in a program. Unrolling is allowed on outer and inner **DO** loops.

Syntax

►► -q unroll nounroll = auto yes ◀◀

Defaults

-qunroll=auto if **-qunroll** is not specified on the command line.

Parameters

- auto** The compiler performs basic loop unrolling.
- yes** The compiler looks for more opportunities to perform loop unrolling than that performed with **-qunroll=auto**. 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.

Usage

Specifying **-qunroll** with no suboptions is equivalent to **-qunroll=yes**.

The **-qnounroll** option prohibits unrolling unless you specify the **STREAM_UNROLL**, **UNROLL**, or **UNROLL_AND_FUSE** directive for a particular loop. These directives always override the command line options.

Examples

In the following example, the **UNROLL(2)** directive is used to tell the compiler that the body of the loop can be replicated so that the work of two iterations is performed in a single iteration. Instead of performing 1000 iterations, if the compiler unrolls the loop, it will only perform 500 iterations.

```
!IBM* UNROLL(2)
      DO I = 1, 1000
         A(I) = I
      END DO
```

If the compiler chooses to unroll the previous loop, the compiler translates the loop so that it is essentially equivalent to the following:

```
      DO I = 1, 1000, 2
         A(I) = I
         A(I+1) = I + 1
      END DO
```

Related information

See the appropriate directive on unrolling loops in the *XL Fortran Language Reference*:

- **STREAM_UNROLL**
- **UNROLL**
- **UNROLL_AND_FUSE**

See *High-order transformation* in the *XL Fortran Optimization and Programming Guide*.

-qunwind

Category

Optimization and tuning

Purpose

Specifies that the compiler will preserve the default behavior for saves and restores to volatile registers during a procedure call.

Syntax

►► — -q —

unwind
nounwind

 —►►

@PROCESS:

@PROCESS UNWIND | NOUNWIND

Defaults

-qunwind

Usage

If you specify **-qnounwind**, the compiler rearranges subprograms to minimize saves and restores to volatile registers. This rearrangement may make it impossible for the program or debuggers to walk through or "unwind" subprogram stack frame chains.

While code semantics are preserved, applications such as exception handlers that rely on the default behavior for saves and restores can produce undefined results. When using **-qnounwind** in conjunction with the **-g** compiler option, debug information regarding exception handling when unwinding the program's stack can be inaccurate.

-qversion

Category

Listings, messages, and compiler information

@PROCESS

None.

Purpose

Displays the version and release of the compiler being invoked.

Syntax



Defaults

-qnoverversion

Parameters

verbose

Additionally displays information about the version, release, and level of each compiler component installed.

Usage

When you specify **-qversion**, the compiler displays the version information and exits; compilation is stopped

-qversion specified without the **verbose** suboption shows compiler information in the format:

```
product_nameVersion: VV.RR.MMMM.LLLL
```

where:

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

Example:

```
IBM XL Fortran for Linux, V13.1  
Version: 13.01.0000.0001
```

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

```
component_name Version: VV.RR(product_name) Level: component_level
```

where:

component_name
Specifies an installed component, such as the low-level optimizer.
component_level
Represents the level of the installed component.

Example:

```
IBM XL Fortran for Linux, V13.1  
Version: 13.01.0000.0000  
Driver Version: 13.01(Fortran) Level: YYMMDD  
Fortran Front End and Run Time Version: 13.01(Fortran) Level: YYMMDD  
Fortran Transformer Version: 13.01(Fortran) Level: YYMMDD  
High-Level Optimizer Version: 11.01(C/C++) and 13.01(Fortran) Level: YYMMDD  
Low-Level Optimizer Version: 11.01(C/C++) and 13.01(Fortran) Level: YYMMDD
```

If you want to save this information to the output object file, you can do so with the **-qsavopt -c** options.

Related information

- “-qsaveopt” on page 230

-qwarn64

Category

Error checking and debugging

@PROCESS

None.

Purpose

Displays informational messages identifying statements that may cause problems with 32-bit to 64-bit migration.

This option aids in porting code from a 32-bit to a 64-bit environment by detecting the truncation of an 8-byte integer pointer to 4 bytes.

Syntax

►► — -q nowarn64
warn64 —————►►

Defaults

-qnowarn64

Usage

You can use the **-qwarn64** option in both 32-bit and 64-bit modes.

The compiler flags the following situations with informational messages:

- The assignment of a reference to the **LOC** intrinsic to an **INTEGER(4)** variable.
- The assignment between an **INTEGER(4)** variable or **INTEGER(4)** constant and an integer pointer.
- The specification of an integer pointer within a common block.
- The specification of an integer pointer within an equivalence statement.

You can use the **-qextchk** option and interface blocks for argument checking.

Related information

- “-q32” on page 112
- “-q64” on page 113
- Chapter 8, “Using XL Fortran in a 64-bit environment,” on page 289

-qxflag=dvz

Category

Error checking and debugging

@PROCESS

None.

Purpose

Causes the compiler to generate code to detect floating-point divide-by-zero operations.

Syntax

►► — -q—xflag— = —dvz— ◀◀

Defaults

Not applicable.

Usage

This option takes effect at optimization levels of **-O** or higher.

With this option on, the extra code calls the external handler function `__xl_dzx` when the divisor is zero. The return value of this function is used as the result of the division. Users are required to provide the function to handle the divide-by-zero operations. Specifying **-qxflag=dvz** handles only single-precision (REAL*4) and double-precision (REAL*8) division.

The interface of the function is as follows:

```
real(8) function __xl_dzx(x, y, kind_type)
  real(8), value :: x, y
  integer, value :: kind_type
end function
```

where:

x is the dividend value

y is the divisor value

kind_type

specifies the size of the actual arguments associated with **x** and **y**.

A **kind_type** value equal to zero indicates that the actual arguments associated with **x** and **y** are of type REAL(8). A **kind_type** value equal to one indicates that the actual arguments associated with **x** and **y** are of type REAL(4).

The division always executes before the handler routine is called. This means that any exception is posted and handled before the handler function is called.

Related information

- *Implementation details of XL Fortran floating-point processing* in the *XL Fortran Optimization and Programming Guide*
- “-qflttrap” on page 158
- “Understanding XL Fortran error messages” on page 291

-qxflag=oldtab

Category

Portability and migration

Purpose

Interprets a tab in columns 1 to 5 as a single character (for fixed source form programs).

Syntax

►► `-qxflag=oldtab` ◀◀

@PROCESS:

@PROCESS XFLAG(OLDTAB)

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.

Usage

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

Category

Language element control

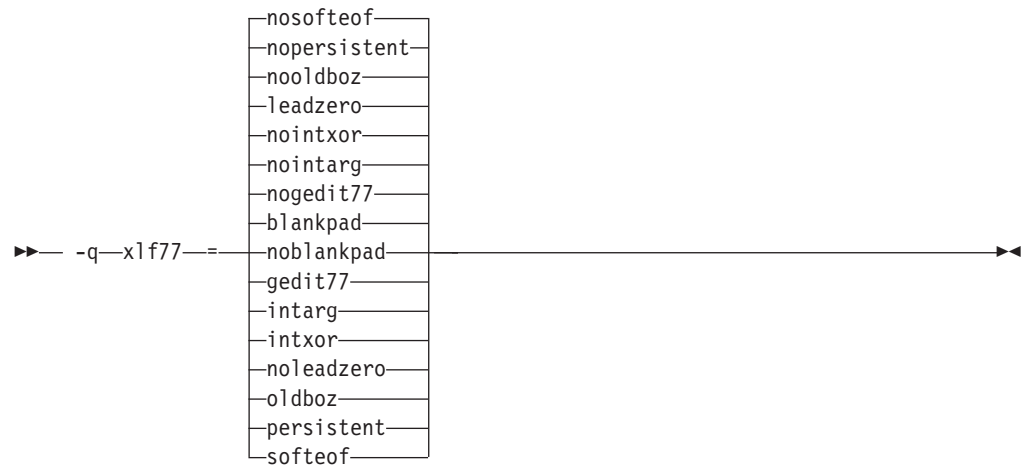
Purpose

Provides compatibility with FORTRAN 77 aspects of language semantics and I/O data format that have changed.

Most of these changes are required by the Fortran 90 standard.

Syntax

Option:



@PROCESS:

@PROCESS XLF77(*settings*)

Defaults

By default, the compiler uses settings that apply to Fortran 95, Fortran 90, Fortran 2003, and the most recent compiler version in all cases.

The default suboptions are: **blankpad**, **nogedit77**, **nointarg**, **nointxor**, **leadzero**, **nooldboz**, **nopersistent**, and **nosofteof**.

These defaults are only used by the **xlf2003**, **xlf2003_r**, **xlf95**, **xlf95_r**, **xlf90**, **xlf90_r**, **f90**, **f95** and **f2003** commands, which you should use to compile new programs.

Parameters

To get various aspects of XL Fortran Version 2 behavior, select the nondefault choice for one or more of the following suboptions. The descriptions explain what happens when you specify the nondefault choices.

blankpad | **noblankpad**

For internal, direct-access, and stream-access files, uses a default setting equivalent to **pad='no'**. This setting produces conversion errors when reading from such a file if the format requires more characters than the record has. This suboption does not affect direct-access or stream-access files opened with a **pad=** specifier.

gedit77 | **nogedit77**

Uses FORTRAN 77 semantics for the output of **REAL** objects with the **G** edit descriptor. Between FORTRAN 77 and Fortran 90, the representation of 0 for a list item in a formatted output statement changed, as did the rounding method, leading to different output for some combinations of values and **G** edit descriptors.

intarg | **nointarg**

Converts all integer arguments of an intrinsic procedure to the kind of the longest argument if they are of different kinds. Under Fortran 90/95 rules, some intrinsics (for example, **IBSET**) determine the result type based on the kind of the first argument; others (for example, **MIN** and **MAX**) require that all arguments be of the same kind.

intxor | **nointxor**

Treats **.XOR.** as a logical binary intrinsic operator. It has a precedence equivalent to the **.EQV.** and **.NEQV.** operators and can be extended with an operator interface. (Because the semantics of **.XOR.** are identical to those of **.NEQV.**, **.XOR.** does not appear in the Fortran 90 or Fortran 95 language standard.)

Otherwise, the **.XOR.** operator is only recognized as a defined operator. The intrinsic operation is not accessible, and the precedence depends on whether the operator is used in a unary or binary context.

leadzero | **noleadzero**

Produces a leading zero in real output under the **D**, **E**, **L**, **F**, and **Q** edit descriptors.

oldboz | **nooldboz**

Turns blanks into zeros for data read by **B**, **O**, and **Z** edit descriptors, regardless of the **BLANK=** specifier or any **BN** or **BZ** control edit descriptors. It also preserves leading zeros and truncation of too-long output, which is not part of the Fortran 90 or Fortran 95 standard.

persistent | **nopersistent**

Saves the addresses of arguments to subprograms with **ENTRY** statements in static storage. This is an implementation choice that has been changed for increased performance.

softeof | **nosofteof**

Allows **READ** and **WRITE** operations when a unit is positioned after its endfile record unless that position is the result of executing an **ENDFILE** statement. This suboption reproduces a FORTRAN 77 extension of earlier versions of XL Fortran that some existing programs rely on.

Usage

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, Fortran 95 and Fortran 2003 and the **xlF90**, **xlF90_r**, **xlF95**, **xlF95_r**, **xlF2003**, **xlF2003_r**, **f90**, **f95** or **f2003** 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.

-qxlf90

Category

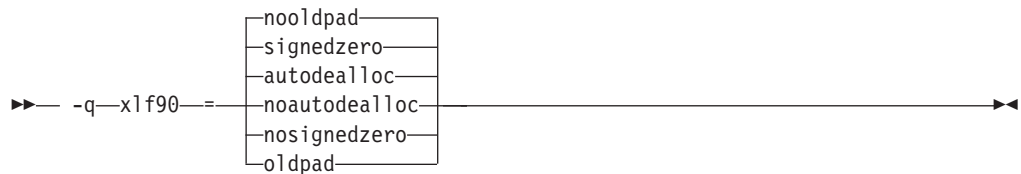
Language element control

Purpose

Provides compatibility with the Fortran 90 standard for certain aspects of the Fortran language.

Syntax

Option:



@PROCESS:

@PROCESS XLF90(*settings*)

Defaults

The default suboptions for **-qxlf90** depend on the invocation command that you specify.

For the **xlf2003**, **f2003**, **xlf2003_r**, **f95**, **xlf95** or **xlf95_r** command, the default suboptions are **signedzero**, **autodealloc**, and **nooldpad**.

For all other invocation commands, the defaults are **nosignedzero**, **noautodealloc** and **oldpad**.

Parameters

signedzero | nosignedzero

Determines how the **SIGN(A,B)** function handles signed real 0.0. If you specify the **-qxlf90=signedzero** compiler option, **SIGN(A,B)** returns **-|A|** when **B=-0.0**. This behavior conforms to the Fortran 95 standard and is consistent with the IEEE standard for binary floating-point arithmetic. Note that for the **REAL(16)** data type, XL Fortran never treats zero as negative zero.

This suboption also determines whether a minus sign is printed in the following cases:

- For a negative zero in formatted output. Again, note that for the **REAL(16)** data type, XL Fortran never treats zero as negative zero.
- For negative values that have an output form of zero (that is, where trailing non-zero digits are truncated from the output so that the resulting output looks like zero). Note that in this case, the **signedzero** suboption does affect the **REAL(16)** data type; non-zero negative values that have an output form of zero will be printed with a minus sign.

When using **-qxlf90=nosignedzero**, consider setting the **-qstrict=nozerosigns** option to improve performance.

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.

oldpad | nooldpad

When the **PAD=specifier** is present in the **INQUIRE** statement, specifying

-qxlf90=nooldpad returns UNDEFINED when there is no connection, or when the connection is for unformatted I/O. This behavior conforms with the Fortran 95 standard and above. Specifying **-qxlf90=oldpad** preserves the Fortran 90 behavior.

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 **-qxlf90** suboption that you specify. For example:

Invocation Command/xlf90 Suboption	Output
xlf2003	-1.0
xlf2003 -qxlf90=signedzero	-1.0
xlf2003 -qxlf90=nosignedzero	1.0
xlf95	-1.0
xlf95 -qxlf90=signedzero	-1.0
xlf95 -qxlf90=nosignedzero	1.0
xlf90	1.0
xlf	1.0

Related information

- See the **SIGN** information in the *Intrinsic Procedures* section and the *Array concepts* section of the *XL Fortran Language Reference*.
- “-qstrict” on page 247

-qxlf2003

Category

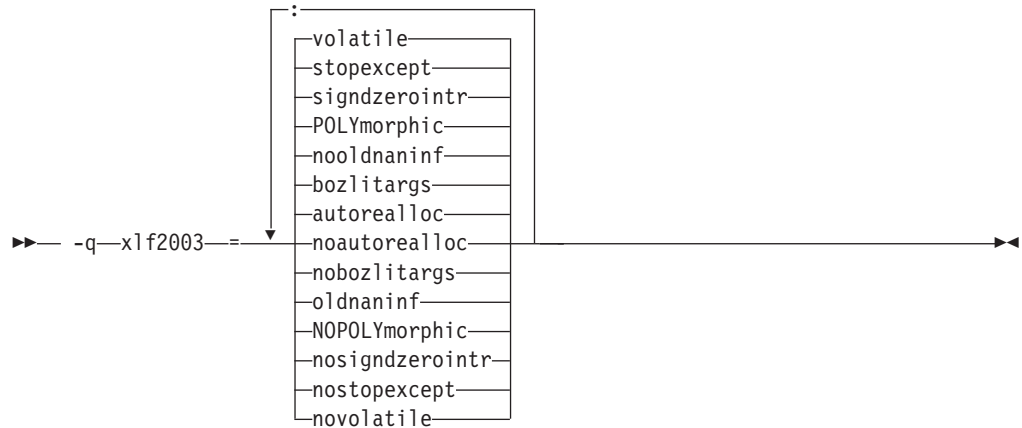
Language element control

Purpose

Provides the ability to use language features specific to the Fortran 2003 standard when compiling with compiler invocations that follow earlier Fortran standards, as well as the ability to disable these features when compiling with compiler invocations that follow the Fortran 2003 standard.

Syntax

Option:



@PROCESS:

@PROCESS XLF2003(*suboption,suboption,...*)

Defaults

The default suboption depends on the invocation command that you specify.

For the **f2003**, **xf2003**, or **xf2003_r** command, the defaults are:

autorealloc:bozlitargs:nooldnaninf:polymorphic:signdzerointr:stopexcept:volatile

For all other invocation commands, the defaults are:

noautorealloc:nobozlitargs:oldnaninf:nopolymorphic:nosigndzerointr:nostopexcept:novolatile

Parameters

autorealloc | noautorealloc

Controls whether the compiler automatically reallocates the left-hand-side (LHS) with the shape of the right-hand-side (RHS) when assigning into an allocatable variable. If the LHS variable was not allocated before the assignment, it is allocated automatically. The default is **autorealloc** for the **f2003**, **xf2003**, and **xf2003_r** commands, and **noautorealloc** for all other commands. This suboption has no effect on reallocation when the values of length type parameters in the LHS and RHS differ.

bozlitargs | nobozlitargs

The **bozlitargs** suboption ensures that the passing of boz-literal constants as arguments to the **INT**, **REAL**, **CMPLX**, or **DBLE** intrinsic function conforms to the Fortran 2003 standard. The default is **bozlitargs** for the **f2003**, **xf2003**, and **xf2003_r** commands. The **-qlanglvl=2003pure** or **-qlanglvl=2003std** option must be specified, as well. If **-qport=typ1ssarg** and **-qxf2003=bozlitargs** are specified, passing boz-literal constants to the **CMPLX** intrinsic will yield non-standard results.

oldnaninf | nooldnaninf

The **oldnaninf** suboption controls the formatting of the output of IEEE NaN and infinity exceptional values. This suboption has no effect on input. When **oldnaninf** is in effect, the compiler uses the XL Fortran V10.1 (and earlier) behavior for output. That is, INF for infinity, NAN for a quiet or signaling NaN.

When **nooldnaninf** is in effect, the compiler output for IEEE exceptional values is compliant with the Fortran 2003 standard. That is, Inf for infinity, NaN(Q) for a quiet NaN, and NaN(S) for a signaling NaN.

polymorphic | nopolymorphic

When **polymorphic** is in effect, the compiler allows polymorphic items in Fortran source files. You can specify the **CLASS** type specifier, the **SELECT TYPE** construct, and use polymorphic items in other Fortran statements. The use of the polymorphic argument also causes the compiler to produce runtime type information for each derived type definition.

When **nopolymorphic** is in effect, polymorphic items cannot be specified in Fortran source files and no runtime type information is generated.

signdzerointr | nosigndzerointr

When **signdzerointr** is in effect, the passing of signed zeros to the **SQRT**, **LOG**, and **ATAN2** intrinsic functions returns results consistent with the Fortran 2003 standard. The **-qxf90=signedzero** option must be in effect, as well. For the **xf**, **xf_r**, **f77**, **fort77**, **xf90**, **xf90_r**, and **f90** invocations, specify both options to have the Fortran 2003 behavior.

The following example shows the use of this suboption:

```
! If the Test program is compiled with -qxf2003=signdzerointr
! and -qxf90=signedzero, then Fortran 2003 behavior is seen.
! Otherwise, this program will demonstrate Fortran 95 behavior.
```

Program Test

```
real a, b
complex j, l
a = -0.0
j = sqrt(cmplx(-1.0,a))
b = atan2(a,-1.0)
l = log(cmplx(-1.0,a))
print *, 'j=', j
print *, 'b=', b
print *, 'l=', l
end
```

! Fortran 95 output:

```
j= (-0.0000000000E+00,1.000000000)
b= 3.141592741
l= (0.0000000000E+00,3.141592741)
```

! Fortran 2003 output:

```
j= (0.0000000000E+00,-1.000000000)
b= -3.141592741
l= (0.0000000000E+00,-3.141592741)
```

stopexcept | nostopexcept

When **stopexcept** is in effect, informational messages are displayed when IEEE floating-point exceptions are signaled by a **STOP** statement. Messages have the format:

```
STOP [stop-code]
(OVERFLOW, DIV-BY-ZERO, INVALID, UNDERFLOW, INEXACT)
```

where *stop-code* corresponds to the optional digit string or character constant specified in the **STOP** statement. OVERFLOW, DIV-BY-ZERO, INVALID, UNDERFLOW and INEXACT appear only if the corresponding flag is set.

The following example shows corresponding messages generated:

```

real :: r11, r12, r13, r14
logical :: l

r11 = 1.3
r12 = 0.0

r13 = r11 / r12 ! divide by zero

r14 = r13 ! to make sure r13 is actually used

r14 = log(-r11) ! invalid input for log

stop "The End"

end

```

Output:

```

STOP The End
(DIV-BY-ZERO, INVALID)

```

When **nostopexcept** is in effect, informational messages are suppressed.

volatile | **novolatile**

When **volatile** is in effect, a non-VOLATILE entity that is use- or host-associated can be specified as VOLATILE in inner or local scope.

Usage

If the application uses F2003 polymorphism, you must compile every unit with **polymorphic** specified . If the application does not use polymorphism, specify the **nopolymorphic** suboption; doing so may save compilation time and potentially improve runtime performance.

Related information

- See the **CLASS** type specifier and the **SELECT TYPE** *construct* in the *XL Fortran Language Reference*.

-qxlines

Category

Input control

Purpose

Specifies whether fixed source form lines with an 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 **-qxlines** 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.

Syntax

Option:



@PROCESS:

@PROCESS XLINES | NOXLINES

Defaults

-qnoxlines

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.

Usage

The conditional compilation characters 'x' and 'd' may be mixed both within a fixed source form program and within a continued source line. If a conditional compilation line is continued onto the next line, all the continuation lines must have 'x' or 'd' in column 1. If the initial line of a continued compilation statement is not a debugging line that begins with either 'x' or 'd' in column 1, subsequent continuation lines may be designated as debug lines as long as the statement is syntactically correct.

The OMP conditional compilation characters '!\$', 'C\$', and '*\$' may be mixed with the conditional characters 'x' and 'd' both in fixed source form and within a continued source line. The rules for OMP conditional characters will still apply in this instance.

Examples

An example of a base case of -qxlines:

```
C2345678901234567890
  program p
    i=3 ; j=4 ; k=5
X   print *,i,j
X   +      ,k
    end program p

<output>: 3 4 5      (if -qxlines is on)
          no output (if -qxlines is off)
```

In this example, conditional compilation characters 'x' and 'd' are mixed, with 'x' on the initial line:

```
C2345678901234567890
  program p
    i=3 ; j=4 ; k=5
X   print *,i,
```

```

D   +      j,
X   +      k
      end program p

```

```

<output>: 3 4 5 (if both -qxlines and -qdlines are on)
          3 5   (if only -qxlines is turned on)

```

Here, conditional compilation characters 'x' and 'd' are mixed, with 'd' on the initial line:

```

C2345678901234567890
      program p
      i=3 ; j=4 ; k=5
D     print *,i,
X   +      j,
D   +      k
      end program p

```

```

<output>: 3 4 5 (if both -qxlines and -qdlines are on)
          3 5   (if only -qdlines is turned on)

```

In this example, the initial line is not a debug line, but the continuation line is interpreted as such, since it has an 'x' in column 1:

```

C2345678901234567890
      program p
      i=3 ; j=4 ; k=5
      print *,i
X   +      ,j
X   +      ,k
      end program p

```

```

<output>: 3 4 5 (if -qxlines is on)
          3     (if -qxlines is off)

```

Related information

- “-D” on page 100
- *Conditional compilation in the XL Fortran Language Reference*

-qxref

Category

Listings, messages, and compiler information

Purpose

Produces a compiler listing that includes the cross-reference component of the attribute and cross-reference section of the listing.

Syntax

```

>>> -q [noxref] [xref] [full] <<<

```

@PROCESS:

@PROCESS XREF[(FULL)] | NOXREF

Defaults

-qnoxref

Usage

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

- “Listings, messages, and compiler information” on page 85
- “Attribute and cross reference section” on page 305

-qzerosize

Category

Optimization and tuning

Purpose

Determines whether checking for zero-sized character strings and arrays takes place in programs that might process such objects.

Syntax

►► -q nozerosize
zerosize ◀◀

@PROCESS:

@PROCESS ZEROSIZE | NOZEROSIZE

Defaults

The default setting depends on which command invokes the compiler: **-qzerosize** for the **xlF90**, **xlF90_r**, **xlF95**, **xlF95_r**, **xlF2003**, **xlF2003_r**, **f90**, **f95**, and **f2003** commands and **-qnozerosize** for the **xlF**, **xlF_r**, and **f77/fort77** commands (for compatibility with FORTRAN 77).

Usage

Use **-qzerosize** for Fortran 90, Fortran 95, and Fortran 2003 programs that might process zero-sized character strings and arrays.

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.

Runtime checking performed by the `-C` option takes slightly longer when `-qzerosize` is in effect.

-r

Category

Object code control

@PROCESS

None.

Purpose

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

Syntax

▶— `-r` —▶

Defaults

Not applicable.

Usage

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

Predefined macros

None.

Examples

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

```
xlf myprogram.f myprog2.f -r -o mytest.o
```

-S

Category

Output control

@PROCESS

None.

Purpose

Generates an assembler language file for each source file.

Syntax

►► -S ◀◀

Rules

When this option is specified, the compiler produces the assembler source files as output instead of an object or an executable file.

Restrictions

The generated assembler files do not include all the data that is included in a `.o` file by `-qipa` or `-g`.

Examples

```
xlf95 -O3 -qhot -S test.f           # Produces test.s
```

Related information

The `-o` option can be used to specify a name for the resulting assembler source file.

-t

Category

Compiler customization

@PROCESS

None.

Purpose

Applies the prefix specified by the `-B` option to the designated components.

Syntax

►► -t a ◀◀

a
b
c
d
F
h
I
l
z

Defaults

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

Parameters

The following table shows the correspondence between **-t** parameters and the component executable names:

Parameter	Description	Executable name
a	Assembler	as
b	Low-level optimizer	xlfcodes
c	Compiler front end	xlfcntry
d	Disassembler	dis
F	C preprocessor	cpp
h	Array language optimizer	xlshot
I	High-level optimizer, compile step	ipa
l	Linker	ld
z	Binder	bolt

Usage

This option is intended to be used together with the **-Bprefix** option.

Note: If you use the **p** suboption, it can cause the source code to be preprocessed separately before compilation, which can change the way a program is compiled.

Examples

To compile `myprogram.f` so that the name `/u/newones/compilers/` is prefixed to the compiler and assembler program names, enter:

```
xlfc myprogram.f -B/u/newones/compilers/ -tca
```

Related information

- “-B” on page 97

-U

Category

Input control

Purpose

Makes the compiler sensitive to the case of letters in names.

Syntax

►► -U ◀◀

@PROCESS:

@PROCESS MIXED | NOMIXED

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.

Usage

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.

If `-U` is specified, case is significant in names. For example, the names `Abc` and `ABC` refer to different objects.

This 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.

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`” on page 200.

-u

Category

Language element control

Purpose

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.

Syntax

▶▶ — `-u` —————▶▶

`@PROCESS:`

`@PROCESS UNDEF | NOUNDEF`

Defaults

`-qnoundef`, which allows implicit typing.

Related information

See **IMPLICIT** in the *XL Fortran Language Reference*.

This is the short form of **-qundef**. See “-qundef” on page 262.

-V

Category

Listings, messages, and compiler information

@PROCESS

None.

Purpose

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

Syntax

▶▶ -V ◀◀

Defaults

Not applicable.

Usage

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

- “-#” on page 96 is similar to **-v**, but it does not actually execute any of the compilation steps.
 - “-V”
-

-V

Category

Listings, messages, and compiler information

@PROCESS

None.

Parameter	Description	Executable name
a	Assembler	as
b	Low-level optimizer	xlfcodes
c	Compiler front end	xlfcntry
d	Disassembler	dis
F	C preprocessor	cpp
h	Array language optimizer	xlfcot
I	High-level optimizer, compile step	ipa
l	Linker	ld
z	Binder	bolt

Usage

In the string following the **-W** option, use a comma as the separator for each option, and do not include any spaces. If you need to include a character that is special to the shell in the option string, precede the character with a backslash. For example, if you use the **-W** option in the configuration file, you can use the escape sequence backslash comma (\,) to represent a comma in the parameter string.

You do not need the **-W** option to pass most options to the linker **ld**; unrecognized command-line options, except **-q** options, are passed to it automatically. Only linker options with the same letters as compiler options, such as **-v** or **-S**, strictly require **-W**.

Examples

To compile the file `file.f` and pass the linker option **-berok** to the linker, enter the following command:

```
xlf -Wl,-berok file.f
```

To compile the file `uses_many_symbols.f` and the assembly file `produces_warnings.s` so that `produces_warnings.s` is assembled with the assembler option **-x** (issue warnings and produce cross-reference), and the object files are linked with the option **-s** (write list of object files and strip final executable file), issue the following command:

```
xlf -Wa,-x -Wl,-s produces_warnings.s uses_many_symbols.f
```

In the following example, the `\,` embeds a literal comma in the **-WF** string and causes three arguments, rather than four, to be supplied to the C preprocessor.

```
$ xlf -qfree=f90 '-WF,-Dint1=1,-Dint2=2,-Dlist=3\,4' a.F
$ cat a.F
print *, int1
print *, int2
print *, list
end
```

The output from the program will be:

```
$ ./a.out
1
2
3 4
```


Defaults

-yn

Parameters

- n Round to nearest.
- m** Round toward minus infinity.
- p** Round toward plus infinity.
- z** Round toward zero..

Usage

If your program contains operations involving real(16) values, the rounding mode must be set to **-yn**, round-to-nearest.

Related information

- “-O” on page 108
- “-qfloat” on page 152
- “-qieee” on page 170

Chapter 8. Using XL Fortran in a 64-bit environment

The 64-bit environment addresses an increasing demand for larger storage requirements and greater processing power. The Linux operating system provides an environment that allows you to develop and execute programs that exploit 64-bit processors through the use of 64-bit address space and 64-bit integers.

To support larger executables that can be fit within a 64-bit address space, a separate, 64-bit object form is used to meet the requirements of 64-bit executables. The linker binds 64-bit objects to create 64-bit executables. Note that objects that are bound together must all be of the same object format. The following scenarios are not permitted and will fail to load, or execute, or both:

- A 64-bit object or executable that has references to symbols from a 32-bit library or shared library
- A 32-bit object or executable that has references to symbols from a 64-bit library or shared library
- A 64-bit executable that attempts to explicitly load a 32-bit module
- A 32-bit executable that attempts to explicitly load a 64-bit module
- Attempts to run 64-bit applications on 32-bit platforms

On both 64-bit and 32-bit platforms, 32-bit executables will continue to run as they currently do on a 32-bit platform. On 32-bit platforms, 64-bit executables can be generated by specifying the **-q64** option.

The XL Fortran compiler mainly provides 64-bit mode support through the **-q64** compiler option in conjunction with the **-qarch** compiler option. This combination determines the bit mode and instruction set for the target architecture. The **-q32** and **-q64** options take precedence over the setting of the **-qarch** option. Conflicts between the **-q32** and **-q64** options are resolved by the "last option wins" rule. Setting **-qarch=ppc** will ensure future compatibility for applications in 32-bit mode. For 64-bit mode applications, use **-qarch=ppc64** to achieve the same effect for all present or future supported 64-bit mode systems. **-qarch** settings that target a specific architecture, like the **pwr5**, **pwr6**, **ppc970**, or **auto** settings will be more system-dependent.

Compiler options for the 64-bit environment

The **-q32**, **-q64**, and **-qwarn64** compiler options are primarily for developers who are targeting 64-bit platforms. They enable you to do the following:

- Develop applications for the 64-bit environment
- Help migrate source code from the 32-bit environment to a 64-bit environment

Chapter 9. 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 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:
 - 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.
 - 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.
- 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 or to improve performance.

Note:

- The message levels **S** and **U** indicate a compilation failure.
- The message levels **I**, **L**, **W**, and **E** do not indicate a compilation failure.

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” on page 254.

Compiler return codes

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 high enough to halt the compilation. Depending on the level of *halt_severity*, 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.

Runtime return codes

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.

Format of XL Fortran diagnostic 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 runtime message from an XL Fortran application program

85 Indicates a loop-transformation message

86 Indicates an interprocedural analysis (IPA) message

87 Indicates a runtime message from the SMP library

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

XL Fortran comes with compiler and runtime messages in U.S. English and Japanese. If compile-time messages are appearing in U.S. English when they should be in another language, verify that the correct message catalogs are installed and that the **LANG**, **LC_MESSAGES**, and/or **LC_ALL** environment variables are set accordingly.

If a runtime message appears in the wrong language, ensure that your program calls the **setlocale** routine to set the program's locale at run time.

To determine which XL Fortran message catalogs are installed, use the following commands to list them:

```
rpm -ql xlf.cmp          # compile-time messages
rpm -ql xlf.msg.rte     # runtime messages
rpm -ql xlsmp.msg.rte   # SMP runtime messages
```

The file names of the message catalogs are the same for all supported international languages, but they are placed in different directories.

Note: When you run an XL Fortran program on a system without the XL Fortran message catalogs, runtime 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.

Related information: See “Environment variables for national language support” on page 8 and “Selecting the language for runtime messages” on page 36.

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:

invocation_command: not found

Message:

```
xlf90: not found
xlf90_r: not found
xlf95: not found
xlf95_r: not found
xlf: not found
xlf_r: not found
xlf2003: not found
xlf2003_r: not found
f77: not found
fort77: not found
f90: not found
f95: not found
f2003: not found
```

System action:

The shell cannot locate the command to execute the compiler.

User response:

Make sure that your **PATH** environment variable includes the directory `/opt/ibmcmp/xlf/13.1/bin`. If the compiler is properly installed, the commands you need to execute it are in this directory.

Could not load program *program*

Message:

```
Could not load program program
Error was: not enough space
```

System action:

The system cannot execute the compiler or an application program at all.

User response:

Set the storage limits for stack and data to “unlimited” for users who experience this problem. For example, you can set both your hard and soft limits with these **bash** commands:

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

Or, you may find it more convenient to edit the file `/etc/security/limits.conf` to give all users unlimited stack and data segments (by entering `-1` for these fields).

If the storage problem is in an XLF-compiled program, using the **-qsave** or **-qsmallstack** option might prevent the program from exceeding the stack limit.

Explanation:

The compiler allocates large internal data areas that may exceed the storage limits for a user. XLF-compiled programs place more data on the

stack by default than in previous versions, also possibly exceeding the storage limit. Because it is difficult to determine precise values for the necessary limits, we recommend making them unlimited.

Could not load library *library_name*

Message:

Could not load program *program*
Could not load library *library_name.so*
Error was: no such file or directory

User response:

Make sure the XL Fortran libraries are installed in `/opt/ibmcmp/xlf/13.1/lib` and `/opt/ibmcmp/xlf/13.1/lib64`, or set the **LD_LIBRARY_PATH** and **LD_RUN_PATH** environment variables to include the directory where **libxlf90.alibxlf90.so** is installed if it is in a different directory. See “Setting library search paths” on page 9 for details of this environment variable.

Messages are displayed in the wrong language

System action:

Messages from the compiler or an XL Fortran application program are displayed in the wrong language.

User response:

Set the appropriate national language environment. You can set the national language for each user with the command **env**. Alternatively, each user can set one or more of the environment variables **LANG**, **NLSPATH**, **LC_MESSAGES**, **LC_TIME**, and **LC_ALL**. If you are not familiar with the purposes of these variables, “Environment variables for national language support” on page 8 provides details.

A compilation fails with an I/O error.

System action:

A compilation fails with an I/O error.

User response:

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.

Too many individual makefiles and compilation scripts

System action:

There are too many individual makefiles and compilation scripts to easily maintain or track.

User response:

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 “Portability and migration” on page 90 for a list of these options and *Porting programs to XL Fortran* in the *XL Fortran Optimization and Programming Guide* for a general discussion of porting.

Isolating problems with individual compilation units

If you find that a particular compilation unit requires specific option settings to compile properly, you may find it more convenient to apply the settings in the source file through an `@PROCESS` directive. Depending on the arrangement of your files, this approach may be simpler than recompiling different files with different command-line options.

Compiling with threadsafe commands

Threadsafe invocation commands like `xlf_r` or `xlf90_r`, for example, use different search paths and call different modules than the non threadsafe invocations. Your programs should account for the different usages. Programs that compile and run successfully for one environment may produce unexpected results when compiled and run for a different use. The configuration file, `xlf.cfg`, shows the paths, libraries, and so on for each invocation command. (See “Editing the default configuration file” on page 14 for an explanation of its contents.)

Running out of machine resources

If the operating system runs low on resources (page space or disk space) while one of the compiler components is running, you should receive one of the following messages:

```
1501-229 Compilation ended because of lack of space.
```

```
1517-011 Compilation ended. No more system resources available.
```

```
1501-053 (S) Too much initialized data.
```

```
1501-511. Compilation failed for file [filename].
```

You may need to increase the system page space and recompile your program. See the man page information `man 8 mkswap swapon` for more information about page space.

If your program produces a large object file, for example, by initializing all or part of a large array, you may need to do one of the following:

- Increase the size of the filesystem that holds the `/tmp` directory.
- Set the `TMPDIR` environment variable to a filesystem with a lot of free space.
- For very large arrays, initialize the array at run time rather than statically (at compile time).

Fixing link-time problems

After the XL Fortran compiler processes the source files, the linker links the resulting object files together. Any messages issued at this stage come from the `ld` command. A frequently encountered error and its solution are listed here for your convenience:

Undefined or unresolved symbols detected

Message:

0706-317 ERROR: Undefined or unresolved symbols detected:

```
filename.o: In function 'main':
filename.o(.text+0x14): undefined reference
to 'p'
filename.o(.text+0x14): relocation truncated
to fit: R_PPC_REL24 p
```

System action:

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.

User response:

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 runtime 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 runtime options” on page 36 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 runtime exceptions are listed in “XL Fortran runtime exceptions” on page 47.

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 “Portability and migration” on page 90 for a list of these options and *Porting programs to XL Fortran* in the *XL Fortran Optimization and Programming Guide* for a general discussion of porting.

Mismatched sizes or types for arguments

Arguments of different sizes or types might produce incorrect execution and results. To do the type-checking during the early stages of compilation, specify interface blocks for the procedures that are called within a program.

Working around problems when optimizing

If you find that a program produces incorrect results when it is optimized and if you can isolate the problem to a particular variable, you might be able to work around the problem temporarily by declaring the variable as **VOLATILE**. This prevents some optimizations that affect the variable. (See **VOLATILE** in the *XL Fortran Language Reference*.) Because this is only a temporary solution, you should continue debugging your code until you resolve your problem, and then remove the **VOLATILE** keyword. If you are confident that the source code and program design are correct and you continue to have problems, contact your support organization to help resolve the problem.

Input/Output errors

If the error detected is an input/output error and you have specified **IOSTAT** on the input/output statement in error, the **IOSTAT** variable is assigned a value according to *Conditions and IOSTAT values* in the *XL Fortran Language Reference*.

If you have installed the XL Fortran runtime message catalog on the system on which the program is executing, a message number and message text are issued to the terminal (standard error) for certain I/O errors. If you have specified **IOMSG** on the input/output statement, the **IOMSG** variable is assigned the error message text if an error is detected, or the content of **IOMSG** variable is not changed. If this catalog is not installed on the system, only the message number appears. Some of the settings in “Setting runtime options” on page 36 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 runtime 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.
END IF
```

See *Installing an exception handler* in the *XL Fortran Optimization and Programming Guide* for instructions about exception handlers and “XL Fortran runtime exceptions” on page 47 for information about the causes of runtime exceptions.

Debugging a Fortran program

You can use **dbx** and other symbolic debuggers to debug your programs. 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.

Note: Debugging Fortran 2003 polymorphic objects and parameterized derived types is not supported in this release of XL Fortran.

Related information:

- “Error checking and debugging” on page 83

Chapter 10. Understanding XL Fortran compiler listings

Diagnostic information is placed in the output listing produced by the **-qlist**, **-qsource**, **-qxref**, **-qattr**, **-qreport**, and **-qlistopt** compiler options. The **-S** option generates an assembler listing in a separate file.

To locate the cause of a problem with the help of a listing, you can refer to the following:

- The source section (to see any compilation errors in the context of the source program)
- The attribute and cross-reference section (to find data objects that are misnamed or used without being declared or to find mismatched parameters)
- The transformation and object sections (to see if the generated code is similar to what you expect)

A heading identifies each major section of the listing. A string of greater than symbols precedes 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 “Listings, messages, and compiler information” on page 85.

Header section

The listing file has a header section that contains the following items:

- A compiler identifier that consists of the following:
 - Compiler name
 - Version number
 - Release number
 - Modification number
 - Fix number
- Source file name
- Date of compilation
- Time of compilation

The header section is always present in a listing; it is the first line and appears only once. The following sections are repeated for each compilation unit when more than one compilation unit is present.

Options section

The options section is always present in a listing. There is a separate section for each compilation unit. It indicates the specified options that are in effect for the compilation unit. This information is useful when you have conflicting options. If you specify the **-qlistopt** compiler option, this section lists the settings for all options.

Source section

The source section contains the input source lines with a line number and, optionally, a file number. The file number indicates the source file (or include file) from which the source line originated. All main file source lines (those that are not from an include file) do not have the file number printed. Each include file has a file number associated with it, and source lines from include files have that file number printed. The file number appears on the left, the line number appears to its right, and the text of the source line is to the right of the line number. XL Fortran numbers lines relative to each file. The source lines and the numbers that are associated with them appear only if the **-qsource** compiler option is in effect. You can selectively print parts of the source by using the **@PROCESS** directives **SOURCE** and **NOSOURCE** throughout the program.

Error messages

If the **-qsource** option is in effect, the error messages are interspersed with the source listing. The error messages that are generated during the compilation process contain the following:

- The source line
- A line of indicators that point to the columns that are in error
- The error message, which consists of the following:
 - The 4-digit component number
 - The number of the error message
 - The severity level of the message
 - The text that describes the error

For example:

```
          2 |          equivalence (i,j,i,j)
            |          .....a.b.
a - "t.f", line 2.24: 1514-117 (E) Same name appears more than once in an equivalence group.
b - "t.f", line 2.26: 1514-117 (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.
```

PDF report section

Additional sections of the listing report have been added to help you understand some aspects of your programs. When using the **-qreport** option with the **-qpdf2** option, you can get the following additional sections added to your listing file in the section entitled PDF Report:

Loop iteration count

The most frequent loop iteration count and the average iteration count, for

a given set of input data, is calculated for most loops in a program. This information is only available when the program is compiled at optimization level -O5.

Block and call count

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

Cache miss

This section of the report is printed in a single table. It reports the number of *Cache Misses* for certain functions, with additional information about the functions such as: Cache Level, Cache Miss Ratio, Line Number, File Name, and Memory Reference.

Note: You must use the option `-qpdf1=level=2` to get this report. You can also select the level of cache to profile using the environment variable `PDF_PM_EVENT` during run time.

For detailed information about the profile-directed feedback, refer to "Profile-directed feedback" in the *XL Fortran Optimization and Programming Guide*.

For additional information about the listing files, see "Understanding XL Fortran compiler listings" in the *XL Fortran Compiler Reference*.

Transformation report section

If the `-qreport` option is in effect, a transformation report listing shows how IBM XL Fortran for Linux, V13.1 optimized the program. This LOOP TRANSFORMATION section displays pseudo-Fortran code that corresponds to the original source code, so that you can see parallelization and loop transformations that the `-qhot` or `-qsmp` options have generated. This section of the report also shows information on the additional transformations and parallelization performed on loop nests if you compile with `-qsmp` and `-qhot=level=2`.

The compiler also reports the number of streams created for a given loop. You can use this information to understand your application code and to tune your code for better performance. For example, you can distribute a loop which has more streams than the number supported by the underlying architecture.

To generate information about where the compiler inserted data prefetch instructions, use the optimization level of `-qhot`, `-O3`, `-O4` or `-O5` together with `-qreport`.

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
   @CIV2 = 0
   Id=1 DO @CIV2 = @CIV2, 24
5|   IF (.FALSE.) GOTO lab_11
   @LoopIV1 = 0
6|   @CSE0 = @CIV2 * 4
   @ICM0 = @CSE0 + 1
   @ICM1 = @CSE0 + 2
   @ICM2 = @CSE0 + 3
   @ICM3 = @CSE0 + 4
5|Id=2 DO @LoopIV1 = @LoopIV1, 99
   ! DIR_INDEPENDENT loopId = 0
   ! DIR_INDEPENDENT loopId = 0
6|   @CSE1 = @LoopIV1 + 1
   SHADV_M003_a(@CSE1,@ICM0) = @ICM0
   SHADV_M002_a(@CSE1,@ICM1) = @ICM1
   SHADV_M001_a(@CSE1,@ICM2) = @ICM2
   SHADV_M000_a(@CSE1,@ICM3) = @ICM3
7|   ENDDO
   lab_11
8|   ENDDO
   lab_9
4|   IF (.FALSE.) THEN
   @LoopIV0 = int((100 - MOD(100, int(4))))
   Id=5 DO @LoopIV0 = @LoopIV0, 100
5|   IF (.FALSE.) GOTO lab_19
   @LoopIV1 = 0
6|   @ICM4 = @LoopIV0 + 1
5|Id=6 DO @LoopIV1 = @LoopIV1, 99
   ! DIR_INDEPENDENT loopId = 0
   ! DIR_INDEPENDENT loopId = 0
6|   a((@LoopIV1 + 1),@ICM4) = @ICM4
7|   ENDDO
   lab_19
8|   ENDDO
   ENDF
9|   END PROGRAM _main

```

Source File	Source Line	Loop Id	Action / Information
0	4	1	Loop interchanging applied to loop nest.
0	4	1	Outer loop has been unrolled 4 time(s).

Data reorganization report section

A summary of useful information about how program variable data gets reorganized by the compiler, *data reorganizations*.

To generate data reorganization information, specify the optimization level **-qipa=level=2** or **-O5** together with **-qreport**. During the IPA link pass, the data reorganization messages for program variable data is produced to the data reorganization section of the listing file. Reorganization include:

- common block splitting
- array splitting
- array transposing
- memory allocation merging
- array interleaving
- array coalescing

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 class of the name
 - The type
 - 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 argument
- Renames
- Static
- Subroutine
- Use associated
- Value parameter

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 number of machine instructions that are produced and the total cycle time (straight-line execution time) 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. The table also includes the file creation date and time.

Compilation unit epilogue Section

This is the last section of the listing for each compilation unit. It contains the diagnostics summary and indicates whether the unit was compiled successfully. This section is not present in the listing if the file contains only one compilation unit.

Compilation epilogue Section

The compilation epilogue section occurs only once at the end of the listing. At completion of the compilation, XL Fortran presents a summary of the compilation: number of source records that were read, compilation start time, compilation end time, total compilation time, total CPU time, virtual CPU time, and a summary of diagnosed conditions. This section is always present in a listing.

Chapter 11. 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.

External names in XL Fortran libraries

To minimize naming conflicts between user-defined names and the names that are defined in the runtime libraries, the names of routines in the runtime libraries are prefixed with an underscore (`_`), or `_xl`.

The XL Fortran runtime 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 Runtime Environment may be divided into these main categories:

- Support for Fortran I/O operations
- Mathematical calculation
- Operating-system services
- Support for SMP parallelization

The XL Fortran Runtime Environment also produces runtime 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 Runtime Environment.

The XL Fortran Runtime Environment is upward-compatible. Programs that are compiled and linked with a given level of the runtime environment and a given level of the operating system require the same or higher levels of both the runtime environment and the operating system to run.

External names in the runtime environment

Runtime 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 runtime subprograms called by Fortran object code.

The names of these runtime subprograms are external symbols. When object code that is produced by the XL Fortran compiler calls a runtime 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 runtime subprogram call with the subprogram definition.

You should avoid using names in your XL Fortran program that conflict with names of runtime subprograms. Conflict can arise under two conditions:

- The name of a subroutine, function, or common block that is defined in a Fortran program has the same name as a library subprogram.

- The Fortran program calls a subroutine or function with the same name as a library subprogram but does not supply a definition for the called subroutine or function.

Technical details of the `-qfloat=hsflt` option

The `-qfloat=hsflt` option is unsafe for optimized programs that compute floating-point values that are outside the range of representation of single precision, not just outside the range of the result type. The range of representation includes both the precision and the exponent range.

Even when you follow the rules that are stated in the preceding paragraph and in “`-qfloat`” on page 152, programs that are sensitive to precision differences might not produce expected results. Because `-qfloat=hsflt` is not compliant with IEEE, programs will not always run as expected.

For example, in the following program, `X.EQ.Y` may be true or may be false:

```

REAL X, Y, A(2)
DOUBLE PRECISION Z
LOGICAL SAME

READ *, Z
X = Z
Y = Z
IF (X.EQ.Y) SAME = .TRUE.
! ...
! ... Calculations that do not change X or Y
! ...
CALL SUB(X)           ! X is stored in memory with truncated fraction.
IF (X.EQ.Y) THEN     ! Result might be different than before.
...

A(1) = Z
X = Z
A(2) = 1.             ! A(1) is stored in memory with truncated fraction.
IF (A(1).EQ.X) THEN ! Result might be different than expected.
...

```

If the value of `Z` has fractional bits that are outside the precision of a single-precision variable, these bits may be preserved in some cases and lost in others. This makes the exact results unpredictable when the double-precision value of `Z` is assigned to single-precision variables. For example, passing the variable as a dummy argument causes its value to be stored in memory with a fraction that is truncated rather than rounded.

Implementation details for `-qautodbl` promotion and padding

The following sections provide additional details about how the `-qautodbl` option works, to allow you to predict what happens during promotion and padding.

Terminology

The *storage relationship* between two data objects determines the relative starting addresses and the relative sizes of the objects. The `-qautodbl` option tries to preserve this relationship as much as possible.

Data objects can also have a *value relationship*, which determines how changes to one object affect another. For example, a program might store a value into one variable, and then read the value through a different storage-associated variable.

With **-qautodbl** in effect, the representation of one or both variables might be different, so the value relationship is not always preserved.

An object that is affected by this option can 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.

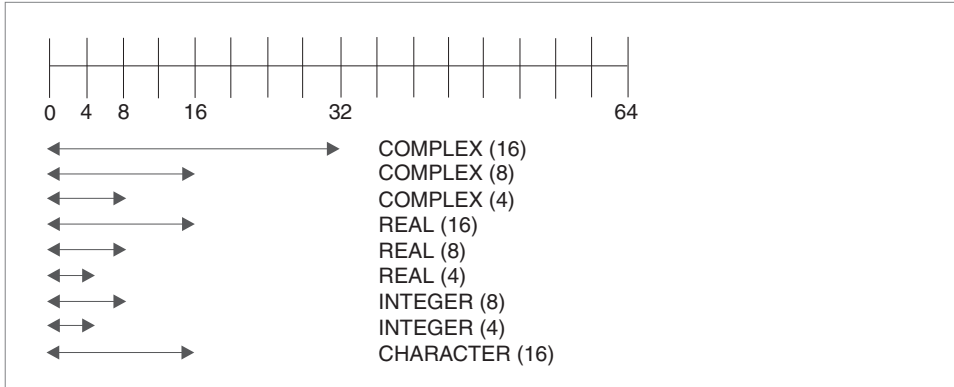


Figure 9. Storage relationships without the `-qautodbl` option

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
```

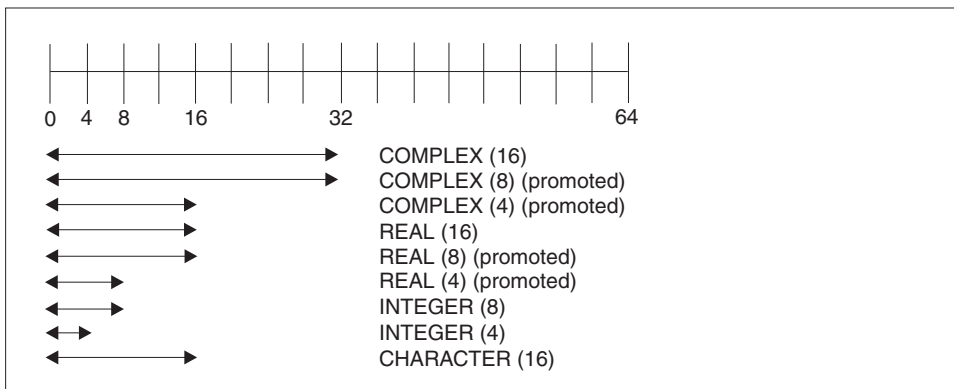


Figure 10. Storage relationships with `-qautodbl=dbl`

```
@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

```

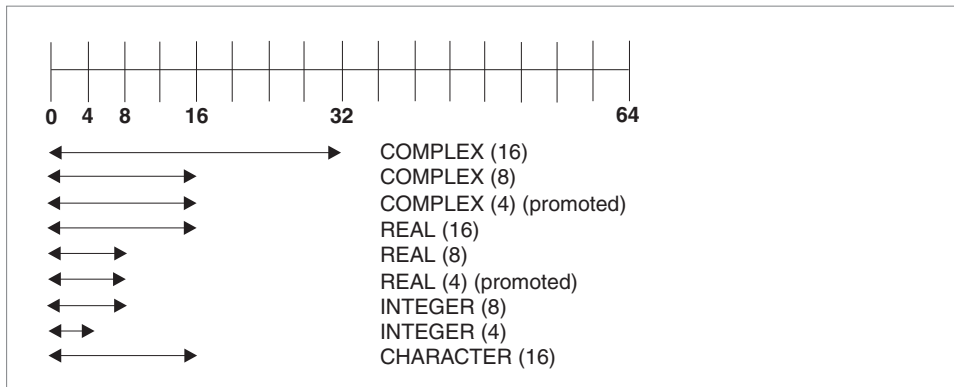


Figure 11. Storage relationships with `-qautobl=dbl4`

```

@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

```

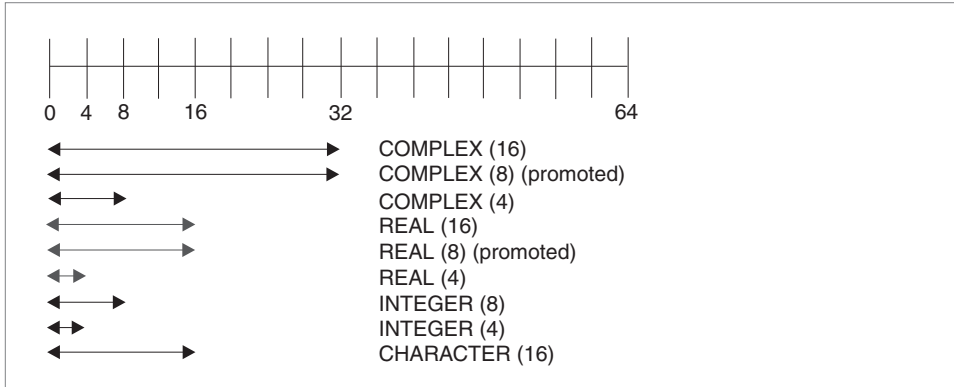


Figure 12. Storage relationships with `-qautodbl=dbl8`

```

@process autodbl(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 occurred
  ! r8(1) = 1.123456789123456789q0      ! promotion did not occur
  ! r8(2) = 2.123456789123456789q0      ! promotion occurred
end

```

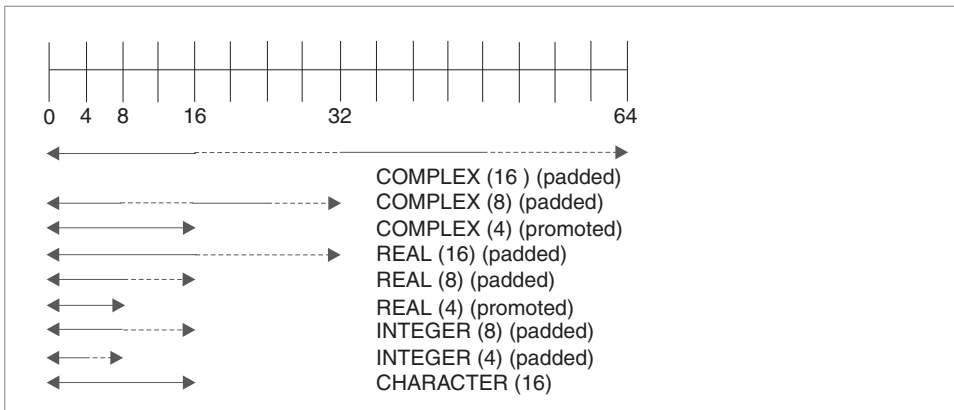


Figure 13. Storage relationships with `-qautodbl=dblpad4`

In the figure above, the dashed lines represent the padding.

```

@process autodbl(db1pad4)
  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

```

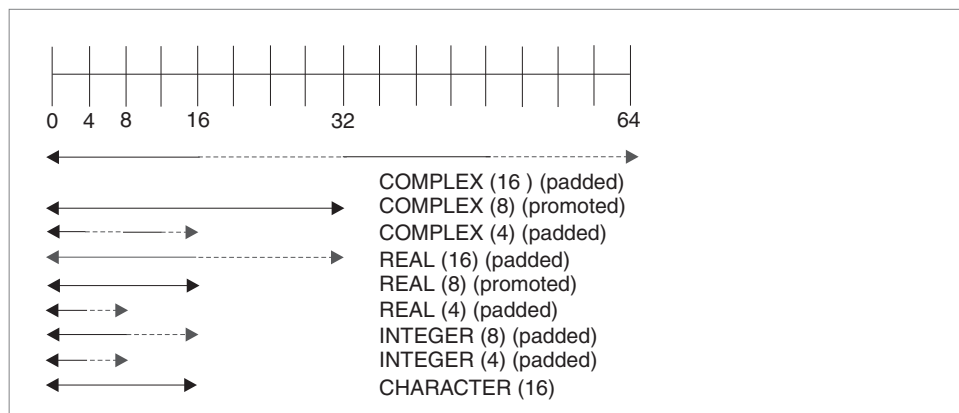


Figure 14. Storage relationships with `-qautodbl=dblpad8`

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)
!
!   x8 = upper 8 bytes of r8(1)                ! promotion occurred
!   r8(1) = real(x16)                          ! padding 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

```

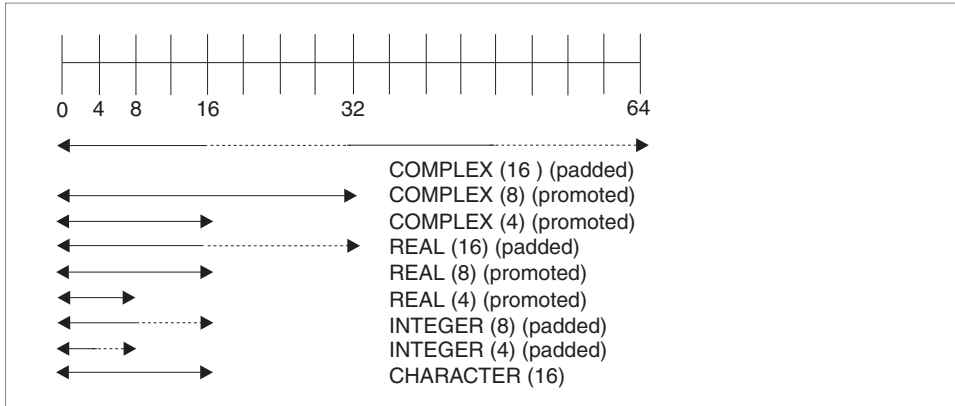


Figure 15. Storage relationships with `-qautodbl=dblpad`

In the figure above, the dashed lines represent the padding.

```
@process autodbl(dblpad)
  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
```

XL Fortran internal limits

Language Feature	Limit
Maximum number of iterations performed by DO loops with loop control with index variable of type INTEGER(<i>n</i>) for <i>n</i> = 1, 2 or 4	$(2^{**31})-1$
Maximum number of iterations performed by DO loops with loop control with index variable of type INTEGER(8)	$(2^{**63})-1$
Maximum character format field width	$(2^{**31})-1$
Maximum length of a format specification	$(2^{**31})-1$
Maximum length of Hollerith and character constant edit descriptors	$(2^{**31})-1$
Maximum length of a fixed source form statement	34 000
Maximum length of a free source form statement	34 000
Maximum number of continuation lines	n/a 1
Maximum number of nested INCLUDE lines	64
Maximum number of nested interface blocks	1 024
Maximum number of statement numbers in a computed GOTO	999
Maximum number of times a format code can be repeated	$(2^{**31})-1$
Allowable record numbers and record lengths for input/output files in 32-bit mode	The record number can be up to $(2^{**63})-1$. The maximum record length is $(2^{**31})-1$ bytes.
Allowable record numbers and record lengths for input/output files in 64-bit mode	The record number can be up to $(2^{**63})-1$, and the record length can be up to $(2^{**63})-1$ bytes. However, for unformatted sequential files, you must use the uwidth=64 runtime option for the record length to be greater than $(2^{**31})-1$ and up to $(2^{**63})-1$. If you use the default uwidth=32 runtime option, the maximum length of a record in an unformatted sequential file is $(2^{**31})-1$ bytes.
Allowable bound range of an array dimension	The bound of an array dimension can be positive, negative, or zero within the range $-(2^{**31})$ to $2^{**31}-1$ in 32-bit mode, or $-(2^{**63})$ to $2^{**63}-1$ in 64-bit mode.
Allowable external unit numbers	0 to $(2^{**31})-1$ 2
Maximum numeric format field width	2 000
Maximum number of concurrent open files	1 024 3

1 You can have as many continuation lines as you need to create a statement with a maximum of 34 000 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 runtime system may open, such as the preconnected units 0, 5, and 6.

Glossary

This glossary defines terms that are commonly used in this document. It includes definitions developed by the American National Standards Institute (ANSI) and entries from the *IBM Terminology* web site.

A

abstract interface

An **ABSTRACT INTERFACE** consists of procedure characteristics and names of dummy arguments. Used to declare the interfaces for procedures and deferred bindings.

abstract type

A type that has the **ABSTRACT** attribute. A nonpolymorphic object cannot be declared to be of abstract type. A polymorphic object cannot be constructed or allocated to have a dynamic type that is abstract.

active processor

See *online processor*.

actual argument

An expression, variable, procedure, or alternate return specifier that is specified in a procedure reference.

alias A single piece of storage that can be accessed through more than a single name. Each name is an alias for that storage.

alphanumeric character

A letter or other symbol, excluding digits, used in a language. Usually the uppercase and lowercase letters A through Z plus other special symbols (such as \$ and _) allowed by a particular language.

alphanumeric

Pertaining to a character set that contains letters, digits, and usually other characters, such as punctuation marks and mathematical symbols.

American National Standard Code for Information Interchange

See *ASCII*.

argument

An expression that is passed to a function or subroutine. See also *actual argument*, *dummy argument*.

argument association

The relationship between an actual argument and a dummy argument during the invocation of a procedure.

arithmetic constant

A constant of type integer, real, or complex.

arithmetic expression

One or more arithmetic operators and arithmetic primaries, the evaluation of which produces a numeric value. An arithmetic expression can be an unsigned arithmetic constant, the name of an arithmetic constant, or a reference to an arithmetic variable, function reference, or a combination of such primaries formed by using arithmetic operators and parentheses.

arithmetic operator

A symbol that directs the performance of an arithmetic operation. The intrinsic arithmetic operators are:

+	addition
-	subtraction
*	multiplication
/	division
**	exponentiation

array An entity that contains an ordered group of scalar data. All objects in an array have the same data type and type parameters.

array declarator

The part of a statement that describes an array used in a program unit. It indicates the name of the array, the number of dimensions it contains, and the size of each dimension.

array element

A single data item in an array, identified by the array name and one or more subscripts. See also *subscript*.

array name

The name of an ordered set of data items.

array section

A subobject that is an array and is not a structure component.

ASCII The standard code, using a coded character set consisting of 7-bit coded characters (8-bits including parity check), that is used for information interchange among data processing systems, data communication systems, and associated equipment. The ASCII set consists of control characters and graphic characters. See also *Unicode*.

asynchronous

Pertaining to events that are not synchronized in time or do not occur in regular or predictable time intervals.

assignment statement

An executable statement that defines or redefines a variable based on the result of expression evaluation.

associate name

The name by which a selector of a **SELECT TYPE** or **ASSOCIATE** construct is known within the construct.

assumed-size array

A dummy array whose size is assumed from the associated actual argument. Its last upper bound is specified by an asterisk.

attribute

A property of a data object that may be specified in a type declaration statement, attribute specification statement, or through a default setting.

automatic parallelization

The process by which the compiler attempts to parallelize both explicitly coded **DO** loops and **DO** loops generated by the compiler for array language.

B

base type

An extensible type that is not an extension of another type.

binary constant

A constant that is made of one or more binary digits (0 and 1).

bind To relate an identifier to another object in a program; for example, to relate an identifier to a value, an address or another identifier, or to associate formal parameters and actual parameters.

binding label

A value of type default character that uniquely identifies how a variable, common block, subroutine, or function is known to a companion processor.

blank common

An unnamed common block.

block data subprogram

A subprogram headed by a **BLOCK DATA** statement and used to initialize variables in named common blocks.

bounds_remapping

Allows a user to view a flat, rank-1 array as a multi-dimensional array.

bss storage

Uninitialized static storage.

busy-wait

The state in which a thread keeps executing in a tight loop looking for more work once it has completed all of its work and there is no new work to do.

byte constant

A named constant that is of type byte.

byte type

A data type representing a one-byte storage area that can be used wherever a **LOGICAL(1)**, **CHARACTER(1)**, or **INTEGER(1)** can be used.

C**character constant**

A string of one or more alphabetic characters enclosed in apostrophes or double quotation marks.

character expression

A character object, a character-valued function reference, or a sequence of them separated by the concatenation operator, with optional parentheses.

character operator

A symbol that represents an operation, such as concatenation (**//**), to be performed on character data.

character set

All the valid characters for a programming language or for a computer system.

character string

A sequence of consecutive characters.

character substring

A contiguous portion of a character string.

character type

A data type that consists of alphanumeric characters. See also *data type*.

- chunk** A subset of consecutive loop iterations.
- class** A set of types comprised of a base type and all types extended from it.
- collating sequence**
The sequence in which the characters are ordered for the purpose of sorting, merging, comparing, and processing indexed data sequentially.
- comment**
A language construct for the inclusion of text in a program that has no effect on the execution of the program.
- common block**
A storage area that may be referred to by a calling program and one or more subprograms.
- compile**
To translate a source program into an executable program (an object program).
- compiler comment directive**
A line in source code that is not a Fortran statement but is recognized and acted on by the compiler.
- compiler directive**
Source code that controls what XL Fortran does rather than what the user program does.
- complex constant**
An ordered pair of real or integer constants separated by a comma and enclosed in parentheses. The first constant of the pair is the real part of the complex number; the second is the imaginary part.
- complex number**
A number consisting of an ordered pair of real numbers, expressible in the form $a+bi$, where a and b are real numbers and i squared equals -1 .
- complex type**
A data type that represents the values of complex numbers. The value is expressed as an ordered pair of real data items separated by a comma and enclosed in parentheses. The first item represents the real part of the complex number; the second represents the imaginary part.
- component**
A constituent of a derived type.
- component order**
The ordering of the components of a derived type that is used for intrinsic formatted input/output and for structure constructors.
- conform**
To adhere to a prevailing standard. An executable program conforms to the Fortran 95 Standard if it uses only those forms and relationships described therein and if the executable program has an interpretation according to the Fortran 95 Standard. A program unit conforms to the Fortran 95 Standard if it can be included in an executable program in a manner that allows the executable program to be standard-conforming. A processor conforms to the standard if it executes standard-conforming programs in a manner that fulfills the interpretations prescribed in the standard.

connected unit

In XL Fortran, a unit that is connected to a file in one of three ways: explicitly via the **OPEN** statement to a named file, implicitly, or by preconnection.

constant

A data object with a value that does not change. The four classes of constants specify numbers (arithmetic), truth values (logical), character data (character), and typeless data (hexadecimal, octal, and binary). See also *variable*.

construct

A sequence of statements starting with a **SELECT CASE**, **DO**, **IF**, or **WHERE** statement, for example, and ending with the corresponding terminal statement.

continuation line

A line that continues a statement beyond its initial line.

control statement

A statement that is used to alter the continuous sequential invocation of statements; a control statement may be a conditional statement, such as **IF**, or an imperative statement, such as **STOP**.

D**data object**

A variable, constant, or subobject of a constant.

data striping

Spreading data across multiple storage devices so that I/O operations can be performed in parallel for better performance. Also known as *disk striping*.

data transfer statement

A **READ**, **WRITE**, or **PRINT** statement.

data type

The properties and internal representation that characterize data and functions. The intrinsic types are integer, real, complex, logical, and character. See also *intrinsic*.

debug line

Allowed only for fixed source form, a line containing source code that is to be used for debugging. Debug lines are defined by a D or X in column 1. The handling of debug lines is controlled by the **-qdlines** and **-qxlines** compiler options.

decimal symbol

The symbol that separates the whole and fractional parts of a real number.

declared type

The type that a data entity is declared to have. May differ from the type during execution (the dynamic type) for polymorphic data entities.

default initialization

The initialization of an object with a value specified as part of a derived type definition.

deferred binding

A binding with the **DEFERRED** attribute. A deferred binding can only appear in an abstract type definition.

definable variable

A variable whose value can be changed by the appearance of its name or designator on the left of an assignment statement.

delimiters

A pair of parentheses or slashes (or both) used to enclose syntactic lists.

denormalized number

An IEEE number with a very small absolute value and lowered precision. A denormalized number is represented by a zero exponent and a non-zero fraction.

derived type

A type whose data have components, each of which is either of intrinsic type or of another derived type.

digit A character that represents a nonnegative integer. For example, any of the numerals from 0 through 9.

directive

A type of comment that provides instructions and information to the compiler.

disk striping

See *data striping*.

DO loop

A range of statements invoked repetitively by a **DO** statement.

DO variable

A variable, specified in a **DO** statement, that is initialized or incremented prior to each occurrence of the statement or statements within a **DO** loop. It is used to control the number of times the statements within the range are executed.

DOUBLE PRECISION constant

A constant of type real with twice the precision of the default real precision.

dummy argument

An entity whose name appears in the parenthesized list following the procedure name in a **FUNCTION**, **SUBROUTINE**, **ENTRY**, or statement function statement.

dynamic dimensioning

The process of re-evaluating the bounds of an array each time the array is referenced.

dynamic extent

For a directive, the lexical extent of the directive and all subprograms called from within the lexical extent.

dynamic type

The type of a data entity during execution of a program. The dynamic type of a data entity that is not polymorphic is the same as its declared type.

E**edit descriptor**

An abbreviated keyword that controls the formatting of integer, real, or complex data.

effective item

A scalar object resulting from expanding an input/output list.

elemental

Pertaining to an intrinsic operation, procedure or assignment that is applied independently to elements of an array or corresponding elements of a set of conformable arrays and scalars.

embedded blank

A blank that is surrounded by any other characters.

entity A general term for any of the following: a program unit, procedure, operator, interface block, common block, external unit, statement function, type, named variable, expression, component of a structure, named constant, statement label, construct, or namelist group.

environment variable

A variable that describes the operating environment of the process.

epoch The starting date used for time in POSIX. It is Jan 01 00:00:00 GMT 1970.

executable program

A program that can be executed as a self-contained procedure. It consists of a main program and, optionally, modules, subprograms and non-Fortran external procedures.

executable statement

A statement that causes an action to be taken by the program; for example, to perform a calculation, test conditions, or alter normal sequential execution.

explicit initialization

The initialization of an object with a value in a data statement initial value list, block data program unit, type declaration statement, or array constructor.

explicit interface

For a procedure referenced in a scoping unit, the property of being an internal procedure, module procedure, intrinsic procedure, external procedure that has an interface block, recursive procedure reference in its own scoping unit, or dummy procedure that has an interface block.

expression

A sequence of operands, operators, and parentheses. It may be a variable, a constant, or a function reference, or it may represent a computation.

extended-precision constant

A processor approximation to the value of a real number that occupies 16 consecutive bytes of storage.

extended type

An extensible type that is an extension of another type. A type that is declared with the **EXTENDS** attribute.

extensible type

A type from which new types may be derived using the **EXTENDS** attribute. A nonsequence type that does not have the **BIND** attribute.

extension type

A base type is an extension type of itself only. An extended type is an extension type of itself and of all types for which its parent type is an extension.

external file

A sequence of records on an input/output device. See also *internal file*.

external name

The name of a common block, subroutine, or other global procedure, which the linker uses to resolve references from one compilation unit to another.

external procedure

A procedure that is defined by an external subprogram or by a means other than Fortran.

F

field An area in a record used to contain a particular category of data.

file A sequence of records. See also *external file*, *internal file*.

file index

See *i-node*.

final subroutine

A subroutine that is called automatically during finalization.

finalizable

A type that has final subroutines, or that has a finalizable component. An object of finalizable type.

finalization

The process of calling user-defined final subroutines immediately before destroying an object.

floating-point number

A real number represented by a pair of distinct numerals. The real number is the product of the fractional part, one of the numerals, and a value obtained by raising the implicit floating-point base to a power indicated by the second numeral.

format

A defined arrangement of such things as characters, fields, and lines, usually used for displays, printouts, or files.

To arrange such things as characters, fields, and lines.

formatted data

Data that is transferred between main storage and an input/output device according to a specified format. See also *list-directed* and *unformatted record*.

function

A procedure that returns the value of a single variable or an object and usually has a single exit. See also *intrinsic procedure*, *subprogram*.

G**generic identifier**

A lexical token that appears in an **INTERFACE** statement and is associated with all the procedures in an interface block.

H**hard limit**

A system resource limit that can only be raised or lowered by using root authority, or cannot be altered because it is inherent in the system or operating environments's implementation. See also *soft limit*.

hexadecimal

Pertaining to a system of numbers to the base sixteen; hexadecimal digits range from 0 (zero) through 9 (nine) and A (ten) through F (fifteen).

hexadecimal constant

A constant, usually starting with special characters, that contains only hexadecimal digits.

high order transformations

A type of optimization that restructures loops and array language.

Hollerith constant

A string of any characters capable of representation by XL Fortran and preceded with *nH*, where *n* is the number of characters in the string.

host

A main program or subprogram that contains an internal procedure is called the host of the internal procedure. A module that contains a module procedure is called the host of the module procedure.

host association

The process by which an internal subprogram, module subprogram, or derived-type definition accesses the entities of its host.

I**IPA**

Interprocedural analysis, a type of optimization that allows optimizations to be performed across procedure boundaries and across calls to procedures in separate source files.

implicit interface

A procedure referenced in a scoping unit other than its own is said to have an implicit interface if the procedure is an external procedure that does not have an interface block, a dummy procedure that does not have an interface block, or a statement function.

implied DO

An indexing specification (similar to a **DO** statement, but without specifying the word **DO**) with a list of data elements, rather than a set of statements, as its range.

infinity

An IEEE number (positive or negative) created by overflow or division by zero. Infinity is represented by an exponent where all the bits are 1's, and a zero fraction.

inherit

To acquire from a parent. Type parameters, components, or procedure bindings of an extended type that are automatically acquired from its parent type without explicit declaration in the extended type are said to be inherited.

inheritance association

The relationship between the inherited components and the parent component in an extended type.

i-node

The internal structure that describes the individual files in the operating system. There is at least one i-node for each file. An i-node contains the node, type, owner, and location of a file. A table of i-nodes is stored near the beginning of a file system. Also known as *file index*.

input/output (I/O)

Pertaining to either input or output, or both.

input/output list

A list of variables in an input or output statement specifying the data to be

read or written. An output list can also contain a constant, an expression involving operators or function references, or an expression enclosed in parentheses.

integer constant

An optionally signed digit string that contains no decimal point.

interface block

A sequence of statements from an **INTERFACE** statement to its corresponding **END INTERFACE** statement.

interface body

A sequence of statements in an interface block from a **FUNCTION** or **SUBROUTINE** statement to its corresponding **END** statement.

interference

A situation in which two iterations within a **DO** loop have dependencies upon one another.

internal file

A sequence of records in internal storage. See also *external file*.

interprocedural analysis

See *IPA*.

intrinsic

Pertaining to types, operations, assignment statements, and procedures that are defined by Fortran language standards and can be used in any scoping unit without further definition or specification.

intrinsic module

A module that is provided by the compiler and is available to any program.

intrinsic procedure

A procedure that is provided by the compiler and is available to any program.

K

keyword

A statement keyword is a word that is part of the syntax of a statement (or directive) and may be used to identify the statement.

An argument keyword specifies the name of a dummy argument

kind type parameter

A parameter whose values label the available kinds of an intrinsic type or a derived-type parameter that is declared to have the **KIND** attribute.

L

lexical extent

All of the code that appears directly within a directive construct.

lexical token

A sequence of characters with an indivisible interpretation.

link-edit

To create a loadable computer program by means of a linker.

linker

A program that resolves cross-references between separately compiled or assembled object modules and then assigns final addresses to create a single relocatable load module. If a single object module is linked, the linker simply makes it relocatable.

list-directed

A predefined input/output format that depends on the type, type parameters, and values of the entities in the data list.

literal A symbol or a quantity in a source program that is itself data, rather than a reference to data.

literal constant

A lexical token that directly represents a scalar value of intrinsic type.

load balancing

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

logical constant

A constant with a value of either true or false (or T or F).

logical operator

A symbol that represents an operation on logical expressions:

.NOT.	(logical negation)
.AND.	(logical conjunction)
.OR.	(logical union)
.EQV.	(logical equivalence)
.NEQV.	(logical nonequivalence)
.XOR.	(logical exclusive disjunction)

loop A statement block that executes repeatedly.

M

_main The default name given to a main program by the compiler if the main program was not named by the programmer.

main program

The first program unit to receive control when a program is run. See also *subprogram*.

master thread

The head process of a group of threads.

module

A program unit that contains or accesses definitions to be accessed by other program units.

mutex A primitive object that provides mutual exclusion between threads. A mutex is used cooperatively between threads to ensure that only one of the cooperating threads is allowed to access shared data or run certain application code at a time.

N**NaN (not-a-number)**

A symbolic entity encoded in floating-point format that does not correspond to a number. See also *quiet NaN*, *signaling NaN*.

name A lexical token consisting of a letter followed by up to 249 alphanumeric characters (letters, digits, and underscores). Note that in FORTRAN 77, this was called a symbolic name.

named common

A separate, named common block consisting of variables.

namelist group name

The first parameter in the NAMELIST statement that names a list of names to be used in READ, WRITE, and PRINT statements.

negative zero

An IEEE representation where the exponent and fraction are both zero, but the sign bit is 1. Negative zero is treated as equal to positive zero.

nest To incorporate a structure or structures of some kind into a structure of the same kind. For example, to nest one loop (the nested loop) within another loop (the nesting loop); to nest one subroutine (the nested subroutine) within another subroutine (the nesting subroutine).

nonexecutable statement

A statement that describes the characteristics of a program unit, data, editing information, or statement functions, but does not cause any action to be taken by the program.

nonexisting file

A file that does not physically exist on any accessible storage medium.

normal

A floating point number that is not denormal, infinity, or NaN.

not-a-number

See *NaN*.

numeric constant

A constant that expresses an integer, real, complex, or byte number.

numeric storage unit

The space occupied by a nonpointer scalar object of type default integer, default real, or default logical.

O

octal Pertaining to a system of numbers to the base eight; the octal digits range from 0 (zero) through 7 (seven).

octal constant

A constant that is made of octal digits.

one-trip DO-loop

A **DO** loop that is executed at least once, if reached, even if the iteration count is equal to 0. (This type of loop is from FORTRAN 66.)

online processor

In a multiprocessor machine, a processor that has been activated (brought online). The number of online processors is less than or equal to the number of physical processors actually installed in the machine. Also known as *active processor*.

operator

A specification of a particular computation involving one or two operands.

P

pad To fill unused positions in a field or character string with dummy data, usually zeros or blanks.

paging space

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

parent component

The component of an entity of extended type that corresponds to its inherited portion.

parent type

The extensible type from which an extended type is derived.

passed-object dummy argument

The dummy argument of a type-bound procedure or procedure pointer component that becomes associated with the object through which the procedure was invoked.

PDF See *profile-directed feedback*.

pointee array

An explicit-shape or assumed-size array that is declared in an integer **POINTER** statement or other specification statement.

pointer

A variable that has the **POINTER** attribute. A pointer must not be referenced or defined unless it is pointer associated with a target. If it is an array, it does not have a shape unless it is pointer-associated.

polymorphic

Able to be of differing types during program execution. An object declared with the **CLASS** keyword is polymorphic.

preconnected file

A file that is connected to a unit at the beginning of execution of the executable program. Standard error, standard input, and standard output are preconnected files (units 0, 5 and 6, respectively).

predefined convention

The implied type and length specification of a data object, based on the initial character of its name when no explicit specification is given. The initial characters I through N imply type integer of length 4; the initial characters A through H, O through Z, \$, and _ imply type real of length 4.

present

A dummy argument is present in an instance of a subprogram if it is associated with an actual argument and the actual argument is a dummy argument that is present in the invoking procedure or is not a dummy argument of the invoking procedure.

primary

The simplest form of an expression: an object, array constructor, structure constructor, function reference, or expression enclosed in parentheses.

procedure

A computation that may be invoked during program execution. It may be a function or a subroutine. It may be an intrinsic procedure, an external procedure, a module procedure, an internal procedure, a dummy procedure, or a statement function. A subprogram may define more than one procedure if it contains **ENTRY** statements.

procedure binding

See type-bound procedure.

procedure pointer

A procedure entity that has the **EXTERNAL** and **POINTER** attributes. It can be pointer associated with an external procedure, a module procedure, a dummy procedure or another procedure pointer.

profile-directed feedback (PDF)

A type of optimization that uses information collected during application execution to improve performance of conditional branches and in frequently executed sections of code.

program unit

A main program or subprogram.

pure An attribute of a procedure that indicates there are no side effects.

Q**quiet NaN**

A NaN (not-a-number) value that does not signal an exception. The intent of a quiet NaN is to propagate a NaN result through subsequent computations. See also *NaN*, *signaling NaN*.

R**random access**

An access method in which records can be read from, written to, or removed from a file in any order. See also *sequential access*.

rank The number of dimensions of an array.

real constant

A string of decimal digits that expresses a real number. A real constant must contain a decimal point, a decimal exponent, or both.

record A sequence of values that is treated as a whole within a file.

relational expression

An expression that consists of an arithmetic or character expression, followed by a relational operator, followed by another arithmetic or character expression.

relational operator

The words or symbols used to express a relational condition or a relational expression:

.GT.	greater than
.GE.	greater than or equal to
.LT.	less than
.LE.	less than or equal to
.EQ.	equal to
.NE.	not equal to

result variable

The variable that returns the value of a function.

return specifier

An argument specified for a statement, such as *CALL*, that indicates to which statement label control should return, depending on the action specified by the subroutine in the *RETURN* statement.

S

scalar A single datum that is not an array.

Not having the property of being an array.

scale factor

A number indicating the location of the decimal point in a real number (and, on input, if there is no exponent, the magnitude of the number).

scope That part of an executable program within which a lexical token has a single interpretation.

scope attribute

That part of an executable program within which a lexical token has a single interpretation of a particular named property or entity.

scoping unit

A derived-type definition.

An interface body, excluding any derived-type definitions and interface bodies contained within it.

A program unit or subprogram, excluding derived-type definitions, interface bodies, and subprograms contained within it.

selector

The object that is associated with the associate name in an **ASSOCIATE** construct.

semantics

The relationships of characters or groups of characters to their meanings, independent of the manner of their interpretation and use. See also *syntax*.

sequential access

An access method in which records are read from, written to, or removed from a file based on the logical order of the records in the file. See also *random access*.

signaling NaN

A NaN (not-a-number) value that signals an invalid operation exception whenever it appears as an operand. The intent of the signaling NaN is to catch program errors, such as using an uninitialized variable. See also *NaN*, *quiet NaN*.

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

SMP See *symmetric multiprocessing*.

soft limit

A system resource limit that is currently in effect for a process. The value of a soft limit can be raised or lowered by a process, without requiring root authority. The soft limit for a resource cannot be raised above the setting of the hard limit. See also *hard limit*.

spill space

The stack space reserved in each subprogram in case there are too many variables to hold in registers and the program needs temporary storage for register contents.

specification statement

A statement that provides information about the data used in the source program. The statement could also supply information to allocate data storage.

stanza A group of lines in a file that together have a common function or define a part of the system. Stanzas are usually separated by blank lines or colons, and each stanza has a name.

statement

A language construct that represents a step in a sequence of actions or a set of declarations. Statements fall into two broad classes: executable and nonexecutable.

statement function

A name, followed by a list of dummy arguments, that is equated with an intrinsic or derived-type expression, and that can be used as a substitute for the expression throughout the program.

statement label

A number made up of one to five digits that is used to identify a statement. Statement labels can be used to transfer control, to define the range of a **DO**, or to refer to a **FORMAT** statement.

storage association

The relationship between two storage sequences if a storage unit of one is the same as a storage unit of the other.

structure

A scalar data object of derived type.

structure component

The part of a data object of derived-type corresponding to a component of its type.

subobject

A portion of a named data object that may be referenced or defined independently of other portions. It can be an array element, array section, structure component, or substring.

subprogram

A function subprogram or a subroutine subprogram. Note that in FORTRAN 77, a block data program unit was called a subprogram. See also *main program*.

subroutine

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

subscript

A subscript quantity or set of subscript quantities enclosed in parentheses and used with an array name to identify a particular array element.

substring

A contiguous portion of a scalar character string. (Although an array section can specify a substring selector, the result is not a substring.)

symmetric multiprocessing (SMP)

A system in which functionally-identical multiple processors are used in parallel, providing simple and efficient load-balancing.

synchronous

Pertaining to an operation that occurs regularly or predictably with regard to the occurrence of a specified event in another process.

syntax The rules for the construction of a statement. See also *semantics*.

T

target A named data object specified to have the **TARGET** attribute, a data object created by an **ALLOCATE** statement for a pointer, or a subobject of such an object.

thread A stream of computer instructions that is in control of a process. A multithread process begins with one stream of instructions (one thread) and may later create other instruction streams to perform tasks.

thread-visible variable

A variable that can be accessed by more than one thread.

time slice

An interval of time on the processing unit allocated for use in performing a task. After the interval has expired, processing unit time is allocated to another task, so a task cannot monopolize processing unit time beyond a fixed limit.

token In a programming language, a character string, in a particular format, that has some defined significance.

trigger constant

A sequence of characters that identifies comment lines as compiler comment directives.

Type-bound procedure

A procedure binding in a type definition. The procedure may be referenced by the binding-name via any object of that dynamic type, as a defined operator, by defined assignment, or as part of the finalization process.

type compatible

All entities are type compatible with other entities of the same type. Unlimited polymorphic entities are type compatible with all entities; other polymorphic entities are type compatible with entities whose dynamic type is an extension type of the polymorphic entity's declared type.

type declaration statement

A statement that specifies the type, length, and attributes of an object or function. Objects can be assigned initial values.

type parameter

A parameter of a data type. **KIND** and **LEN** are the type parameters of intrinsic types. A type parameter of a derived type has either a **KIND** or a **LEN** attribute.

Note: The type parameters of a derived type are defined in the derived-type definition.

U**unformatted record**

A record that is transmitted unchanged between internal and external storage.

Unicode

A universal character encoding standard that supports the interchange, processing, and display of text that is written in any of the languages of the modern world. It also supports many classical and historical texts in a number of languages. The Unicode standard has a 16-bit international character set defined by ISO 10646. See also *ASCII*.

unit A means of referring to a file to use in input/output statements. A unit can be connected or not connected to a file. If connected, it refers to a file. The connection is symmetric: that is, if a unit is connected to a file, the file is connected to the unit.

unsafe option

Any option that could result in undesirable results if used in the incorrect context. Other options may result in very small variations from the default

result, which is usually acceptable. Typically, using an unsafe option is an assertion that your code is not subject to the conditions that make the option unsafe.

use association

The association of names in different scoping units specified by a **USE** statement.

V

variable

A data object whose value can be defined and redefined during the execution of an executable program. It may be a named data object, array element, array section, structure component, or substring. Note that in FORTRAN 77, a variable was always scalar and named.

X

XPG4 X/Open Common Applications Environment (CAE) Portability Guide Issue 4; a document which defines the interfaces of the X/Open Common Applications Environment that is a superset of POSIX.1-1990, POSIX.2-1992, and POSIX.2a-1992 containing extensions to POSIX standards from XPG3.

Z

zero-length character

A character object that has a length of 0 and is always defined.

zero-sized array

An array that has a lower bound that is greater than its corresponding upper bound. The array is always defined.

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:

Lab Director
IBM Canada Ltd. Laboratory
8200 Warden Avenue
Markham, Ontario L6G 1C7
Canada

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. 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.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. 1998, 2010. All rights reserved.

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.

Trademarks and service marks

IBM, the IBM logo, and `ibm.com` are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at <http://www.ibm.com/legal/copytrade.shtml>.

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

Cell Broadband Engine is a trademark of Sony Computer Entertainment, Inc. in the United States, other countries, or both and is used under license therefrom.

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

Other company, product, or service names may be trademarks or service marks of others.

Index

Special characters

`_OPENMP` C preprocessor macro 32

`-#` compiler option 96
`-1` compiler option 97
`-B` compiler option 97
`-c` compiler option 99
`-C` compiler option 98
`-d` compiler option 100
`-D` compiler option 100
`-F` compiler option 102
`-g` compiler option 103, 299
`-I` compiler option 104
`-k` compiler option 105
`-l` compiler option 106
`-L` compiler option 105
`-NS` compiler option 107
`-o` compiler option 110
`-O` compiler option 108
`-O2` compiler option 108
`-O3` compiler option 108
`-O4` compiler option 109
`-O5` compiler option 109
`-p` compiler option 111
`-q32` compiler option 112
`-q64` compiler option 113
`-qalias` compiler option 114
`-qalias_size` compiler option 117
`-qalign` compiler option 118
`-qarch` compiler option 31
`-qassert` compiler option 124
`-qattr` compiler option 125, 305
`-qautodbl` compiler option 126, 310
`-qbindcextname` compiler option 128
`-qcache` compiler option 31, 129
`-qcclines` compiler option 132
`-qcheck` compiler option 98, 132
`-qci` compiler option 133
`-qcompact` compiler option 134
`-qcr` compiler option 134
`-qctypass` compiler option 135
`-qdbg` compiler option 103, 136
`-qddim` compiler option 137
`-qdescriptor` compiler option 138
`-qdirective` compiler option 139
`-qdlines` compiler option 100, 142
`-qdpccomp` compiler option 142
`-qenum` compiler option 143
`-qescape` compiler option 144
`-qessl` compiler option 146
`-qextern` compiler option 147
`-qextname` compiler option 148
`-qfdpr` compiler option 149
`-qfixed` compiler option 150
`-qflag` compiler option 151
`-qfltrap` compiler option 158
`-qfpp` option 157
`-qfree` compiler option 159
`-qfullpath` compiler option 161
`-qfunctrace` 162
`-qfunctrace_xlf_catch` 163
`-qfunctrace_xlf_enter` 164
`-qfunctrace_xlf_exit` 165
`-qhalt` compiler option 166
`-qieee` compiler option 286
`-qinit` compiler option 172
`-qinitauto` compiler option 172
`-qinlgue` compiler option 175
`-qinline` 176
`-qintlog` compiler option 178
`-qintsize` compiler option 179
`-qkepparm` compiler option 187
`-qlanglvl` compiler option 187
`-qlibansi` compiler option 189
`-qlibmpi` 190
`-qlinedebug` compiler option 191
`-qlist` compiler option 192, 306
`-qlistfmt` compiler option 193
`-qlistopt` compiler option 195, 301
`-qlog4` compiler option 196
`-qmaxmem` compiler option 197
`-qmbcs` compiler option 198
`-qminimaltoc` compiler option 199
`-qmixed` compiler option 200
`-qmkshobj` 24
`-qmoddir` compiler option 201
`-qmodule` compiler option 202
`-qnofunctrace` 162
`-qnoinline` 176
`-qnoprint` compiler option 203
`-qnullterm` compiler option 204
`-qobject` compiler option 205
`-qonetrup` compiler option 97, 206
`-qoptdebug` compiler option 206
`-qoptimize` compiler option 108, 207
`-qphsinfo` compiler option 213
`-qpic` compiler option 215
`-qport` compiler option 216
`-qposition` compiler option 218
`-qppsuborigarg` option 219
`-qqcount` compiler option 222
`-qrealize` compiler option 223
`-qrecur` compiler option 225
`-qreport` compiler option 226, 302, 303, 305
`-qsaa` compiler option 228
`-qsave` compiler option 229
`-qsaveopt` compiler option 230
`-qsclock` compiler option 232
`-qshowpdf` compiler option 233
`-qsigtrap` compiler option 234
`-qsmallstack` compiler option 236
`-qsmpp` compiler option 237
`-qsource` compiler option 242, 302
`-qspillsize` compiler option 107, 243
`-qstackprotect` compiler option 244
`-qstacktemp` compiler option 245
`-qstrict` compiler option 247
`-qstrict_induction` compiler option 252
`-qstrictieemod` compiler option 251
`-qsuffix` compiler option 253
`-qsuppress` compiler option 254
`-qswapomp` compiler option 256

`-qhtable` compiler option 257
`-qthreaded` compiler option 258
`-qtimestamps` compiler option 259
`-qtune` compiler option 31
`-qundef` compiler option 262, 282
`-qunroll` compiler option 262
`-qunwind` compiler option 264
`-qversion` compiler option 264
`-qwasm64` compiler option 266
`-qxflag=dvz` compiler option 266
`-qxflag=oldtab` compiler option 268
`-qxf2003` compiler option 272
`-qxf77` compiler option 268
`-qxf90` compiler option 270
`-qxlines` compiler option 275
`-qxref` compiler option 277, 305
`-qzerosize` compiler option 278
`-S` compiler option 279
`-u` compiler option 282
`-U` compiler option 281
`-v` compiler option 283
`-V` compiler option 283
`-w` compiler option 151, 286
`-yn, -ym, -yp, -yz` compiler options 170, 286
`/tmp` directory 10
`.a` files 25
`.cfg` files 25
`.f` and `.F` files 25
`.lst` files 27
`.mod` file names 202
`.mod` files 25, 27, 201
`.o` files 25, 27
`.s` files 25, 27
`.so` files 25
`.XOR` operator 269
`@PROCESS` compiler directive 29
`#if` and other `cpp` directives 33
`#pragma nofunctrace` 162

Numerics

1501-224, 1501-229, and 1517-011 error messages 296
15xx identifiers for XL Fortran messages 292
4K suboption of `-qalign` 118
64-bit environment 289

A

`a.out` file 27
actual arguments
 definition of 319
addresses of arguments, saving 269
aggressive array I/O 37
aggressive_array_io runtime option 37
`ALIAS @PROCESS` directive 114
alias table size 117
`ALIGN @PROCESS` directive 118

alignment of BIND(C) derived types 118
 alignment of CSECTs and large arrays for data-striped I/O 118
 allocatable arrays, automatic deallocation with `-qxlF90=autodealloc` 270
 alphabetic character, definition of 319
 alphanumeric, definition of 319
 alter program semantics 247
 ANSI
 checking conformance to the Fortran 90 standard 4, 42, 187
 checking conformance to the Fortran 95 standard 4, 42, 187
 appendold and appendunknown suboptions of `-qposition` 218
 architecture 120
 `-qarch` compiler option 120
 `-qtune` compiler option 260
 architecture combination 261
 archive files 25
 argument addresses, saving 269
 argument promotion (integer only) for intrinsic procedures 269
 arguments
 definition of 319
 passing null-terminated strings to C functions 204
 arrays
 optimizing assignments 114
 padding 167
 arrays, initialization problems 296
 aryovrlp suboption of `-qalias` 114
 as attribute of configuration file 15
 as command, passing command-line options to 30
 ASCII
 definition of 320
 asopt attribute of configuration file 15
 assembler
 source (.s) files 25, 27
 ATTR @PROCESS directive 125
 attribute section in compiler listing 305
 AUTODBL @PROCESS directive 126
 autodealloc suboption of `-qxlF90` 270
 autorealloc suboption, `-qxlF2003` 273

B

backward 17
 Backward compatibility issues 17
 bash shell 8
 basic example, described xii
 BIND(C) derived types, alignment 118
 BINDCEXTNAME @PROCESS directive 128
 blankpad suboption of `-qxlF77` 269
 bolt attribute of configuration file 16
 bozlitargs suboption, `-qxlF2003` 273
 bss storage, alignment of arrays in 118
 buffering runtime option
 description 37
 using with preconnected files 37

C

C preprocessor (cpp) 31, 157, 219
 carriage return character 134
 catch routine 163
 CCLINES @PROCESS 132
 character constants and typeless constants 135
 CHECK @PROCESS directive 98, 132
 chunk
 definition of 322
 CI @PROCESS directive 133
 cleanpdf command 210
 cnverr runtime option 39
 code attribute of configuration file 15
 code generation for different systems 31
 code optimization 6
 command line options, summary 77
 command line, specifying options on 28
 COMPACT @PROCESS directive 134
 compatibility 17
 compatibility options for compatibility 90
 compilation order 25
 compilation unit epilogue section in compiler listing 306
 compiler listings 301
 compiler options
 deprecated 92
 descriptions 95
 obsolete or not recommended 92
 scope and precedence 28
 section in compiler listing 301
 specifying in the source file 29
 specifying on the command line 28
 compiler options for 64-bit 289
 compiling
 cancelling a compilation 25
 description of how to compile a program 21
 Fortran 2003 programs 23
 problems 296
 SMP programs 25
 conditional compilation 31
 configuration 10
 custom configuration files 10
 configuration file 25, 102
 configuration file attributes 14
 conflicting options
 `-C` interferes with `-qhot` 99
 `-qautodbl` overrides `-qrealsize` 128
 `-qdpC` is overridden by `-qautodbl` and `-qrealsize` 223
 `-qflag` overrides `-qlanglvl` or `-qsaA` 152
 `-qhalt` is overridden by `-qnoobject` 206
 `-qhalt` overrides `-qobject` 206
 `-qintsize` overrides `-qlog4` 197
 `-qlanglvl` is overridden by `-qflag` 188
 `-qlog4` is overridden by `-qintsize` 197
 `-qnoobject` overrides `-qhalt` 167
 `-qobject` is overridden by `-qhalt` 167
 `-qrealsize` is overridden by `-qautodbl` 128, 223
 `-qrealsize` overrides `-qdpC` 223
 `-qsaA` is overridden by `-qflag` 228

conflicting options (*continued*)
 `@PROCESS` overrides command-line setting 28
 command-line overrides configuration file setting 28
 specified more than once, last one takes effect 28
 conformance checking 4, 187, 228
 control of implicit timestamps 259
 control of transformations 247
 control size of alias table 117
 conversion errors 39
 core file 298
 could not load program (error message) 294
 cpp attribute of configuration file 15
 cpp command 31
 cpp, cppoptions, and cppsuffix attributes of configuration file 15
 cppsuffix attribute of configuration file 16
 cpu_time_type runtime option 40
 cross reference section in compiler listing 305
 crt attribute of configuration file 14
 crt_64 attribute of configuration file 14
 CSECTs, alignment of 118
 csh shell 8
 CTYPLSS @PROCESS directive 135
 customizing configuration file (including default compiler options) 14

D

data limit 294
 data reorganization report section in compiler listing 305
 data striping
 `-qalign` required for improved performance 118
 DBG @PROCESS directive 103, 136
 dbl, db14, db18, dblpad, dblpad4, dblpad8 suboptions of `-qautodbl` 126
 DDIM @PROCESS directive 137
 debug optimized code 206
 debugger support 6
 debugging 291
 using path names of original files 161
 default_recl runtime option 40
 defaultmsg attribute of configuration file 16
 defaults
 customizing compiler defaults 14
 search paths for include and .mod files 104
 search paths for libraries 9
 deprecated compiler options 92
 deps suboption of `-qassert` 124
 descriptor data structure formats 138
 diagnostics, compiler listings 301
 DIRECTIVE @PROCESS directive 139
 disassembly listing
 from the `-S` compiler option 279
 disk space, running out of 296
 DLINEs @PROCESS directive 100, 142
 DPC @PROCESS directive 142

dummy argument
 definition of 324

dynamic dimensioning of arrays 137

dynamic extent, definition of 324

dynamic library 24

dynamic linking 34

E

E error severity 291

edit descriptors (B, O, Z), differences
 between F77 and F90 269

edit descriptors (G), difference between
 F77 and F90 269

editing configuration file 14

editing source files 21

emacs text editor 21

enable suboption of -qfltrap 158

end-of-file, writing past 269

enter routine 164

ENTRY statements, compatibility with
 previous compiler versions 269

environment problems 294

environment variables
 compile time 8
 LANG 8
 NLSPATH 8
 PDF_BIND_PROCESSOR 9
 TMPDIR 10
 LD_LIBRARY_PATH 9
 LD_RUN_PATH 9
 runtime 9
 LD_LIBRARY_PATH 47
 LD_RUN_PATH 47
 TMPDIR 47
 XLFRTFOPTS 36
 XLF_USR_CONFIG 47
 XLFSCRATCH_unit 10
 XLFUNIT_unit 10

eof, writing past 269

epilogue sections in compiler listing 306

err_recovery runtime option 41

errloc runtime option 40

error checking and debugging 83

error messages 291
 1501-229 296
 1517-011 296
 explanation of format 292
 in compiler listing 302

erroeof runtime option 41

errthrdnum runtime option 41

ESCAPE @PROCESS directive 144

exception handling 47
 for floating point 158

exclusive or operator 269

executable files 27

executing a program 35

executing the compiler 21

exit routine 165

extensions, language 3

external names
 in the runtime environment 309

EXTNAME @PROCESS directive 148

F

faster array I/O 37

file table section in compiler listing 306

files
 editing source 21
 input 25
 output 27
 using suffixes other than .f for source
 files 16

FIPS FORTRAN standard, checking
 conformance to 4

FIXED @PROCESS directive 150

FLAG @PROCESS directive 151

floating-point
 exception handling 47
 exceptions 158

FLTRAP @PROCESS directive 158

Fortran 2003
 programs, compiling 23

Fortran 2003 features 42

Fortran 2003 iostat_end behavior 41

Fortran 90
 compiling programs written for 23

Fortran 95
 compiling programs written for 23

FREE @PROCESS directive 159

fsuffix attribute of configuration file 16

full suboption of -qbttable 257

FULLPATH @PROCESS directive 161

function trace 162, 163, 164, 165

functrace 162, 163, 164, 165

G

G edit descriptor, difference between F77
 and F90 269

gcc_libs attribute of configuration file 14

gcc_libs_64 attribute of configuration
 file 15

gcc_path attribute of configuration
 file 15

gcc_path_64 attribute of configuration
 file 15

gcr4 attribute of configuration file 14

gcr464 attribute of configuration file 14

gedit77 suboption of -qxlf77 269

generating code for different systems 31

H

HALT @PROCESS directive 166

hardware, compiling for different types
 of 31

header section in compiler listing 301

hexint and nohexint suboptions of
 -qport 216

high order transformation 167

hot attribute of configuration file 16

hsflt suboption of -qfloat 310

I

I error severity 291

i-node 43

IEEE @PROCESS directive 170, 286

IEEE infinity output 273

IEEE NaN output 273

implicit timestamps, control of 259

imprecise suboption of -qfltrap 158

include attribute of configuration file 16
 include_32 attribute of configuration
 file 16

include_64 attribute of configuration
 file 16

inexact suboption of -qfltrap 158

informational message 291

INIT @PROCESS directive 172

initialize arrays, problems 296

INLGLUE @PROCESS directive 175

inlining 176

input files 25

input/output
 increasing throughput with data
 striping 118
 runtime behavior 36
 when unit is positioned at
 end-of-file 269

installation problems 294

installing the compiler 7

intarg suboption of -qxlf77 269

integer arguments of different kinds to
 intrinsic procedures 269

internal limits for the compiler 317

interprocedural analysis (IPA) 181

INTLOG @PROCESS directive 178

intptr suboption of -qalias 114

intrinsic procedures accepting integer
 arguments of different kinds 269

intrinths runtime option 41

INTSIZE @PROCESS directive 179

intxor suboption of -qxlf77 269

invalid suboption of -qfltrap 158

invoking a program 35

invoking the compiler 21

iostat_end runtime option 41

ipa attribute of configuration file 16

irand routine, naming restriction for 34

ISO
 checking conformance to the Fortran
 2003 standard 4
 checking conformance to the Fortran
 90 standard 4, 42, 187
 checking conformance to the Fortran
 95 standard 4, 42, 187

itercnt suboption of -qassert 124

K

kind type parameters 180, 224

ksh shell 8

L

L error severity 291

LANG environment variable 8

LANGLVL @PROCESS directive 187

langlvl runtime option 42

language extensions 3

language standards 3

language support 3

language-level error 291

LC_* national language categories 9
 ld command, passing command-line options to 30
 LD_LIBRARY_PATH environment variable 9, 47
 LD_RUN_PATH environment variable 9, 47
 ldopt attributes of configuration file 15
 leadzero suboption of -qxf77 269
 level of XL Fortran, determining 16
 lexical extent, definition of 328
 lib*.so library files 25, 106
 libraries 25
 default search paths 9
 shared 309
 libraries attribute of configuration file 16
 library
 shared (dynamic) 24
 static 24
 library path environment variable 294
 libxlf90_t.so 22
 libxlf90.so library 36
 libxlnp.so library 36
 limit command 294
 limits internal to the compiler 317
 line feed character 134
 LINEDEBUG @PROCESS directive 191
 linking 33
 dynamic 34
 problems 297
 static 34
 LIST @PROCESS directive 192
 listing files 27
 listings, compiler 301
 LISTOPT @PROCESS directive 195
 little-endian I/O 45
 locale, setting at run time 36
 LOG4 @PROCESS directive 196

M

m suboption of -y 287
 machines, compiling for different types 31, 120
 macro expansion 31, 157, 219
 macro, _OPENMP C preprocessor 32
 maf suboption of -qfloat 250
 make command 96
 makefiles
 copying modified configuration files along with 14
 malloc system routine 128
 MAXMEM @PROCESS directive 197
 MBCS @PROCESS directive 198
 mclck routine, naming restrictions for 34
 mcrt64 attribute of configuration file 14
 mergepdf 210
 message suppression 254
 messages
 1501-053 error message 296
 1501-229 error message 296
 1517-011 error message 296
 catalog files for 293
 copying message catalogs to another system 293

messages (*continued*)
 selecting the language for runtime messages 36
 migrating 4
 minus suboption of -qieee 170
 MIXED @PROCESS directive 200, 281
 mod and nomod suboptions of
 -qport 216
 mod file names, intrinsic 202
 mod files 25, 27, 201
 modules, effect on compilation order 25
 mon.out file 25
 mpi 190
 MPI 190
 multconn runtime option 43
 multconnio runtime option 43

N

n suboption of -y 287
 name conflicts, avoiding 34
 namelist runtime option 44
 NaN values
 specifying with -qinitauto compiler option 172
 naninfoutput runtime option 44
 national language support
 at run time 36
 compile time environment 8
 nearest suboption of -qieee 170
 network file system (NFS)
 using the compiler on a 7
 Network Install Manager 7
 NFS 7
 NIM (Network Install Manager) 7
 NLSPATH environment variable
 compile time 8
 nlwidth runtime option 44
 nodblpad suboption of -qautodbl 126
 nodeps suboption of -qassert 124
 NOFUNCTRACE 162
 none suboption of -qautodbl 126
 none suboption of -qtbltable 257
 nooldnaninf suboption, -qxf2003 273
 nooldpad suboption of -qxf90 270
 null-terminated strings, passing to C functions
 strings, passing to C functions 204
 NULLTERM @PROCESS directive 204

O

OBJECT @PROCESS directive 205
 object files 25, 27
 object output, implicit timestamps 259
 obsolete compiler options 92
 oldboz suboption of -qxf77 269
 oldpad suboption of -qxf90 270
 ONETRIP @PROCESS directive 97, 206
 optimization 6
 loop optimization 167
 OPTIMIZE @PROCESS directive 108, 207
 options attribute of configuration file 15
 options for performance optimization 86
 options section in compiler listing 301

options that control linking 90
 options that control listings and messages 85
 osuffix attribute of configuration file 16
 output files 27
 overflow suboption of -qflttrap 159

P

p suboption of -y 287
 pad setting, changing for internal, direct-access and stream-access files 269
 padding of data types with -qautodbl option 311
 paging space
 running out of 296
 path name of source files, preserving with -qfullpath 161
 PDF_BIND_PROCESSOR environment variable 9
 PDF_PM_EVENT 9
 PDF_PM_EVENT environment variables 9
 PDFDIR 9
 PDFDIR environment variables 9
 PDFreport section in compiler listing 302
 performance of real operations, speeding up 127, 223
 performance optimization options 86
 persistent suboption of -qxf77 269
 PHSINFO @PROCESS directive 213
 platform, compiling for a specific type 120
 plus suboption of -qieee 170
 pointers (Fortran 90) and -qinit compiler option 172
 polymorphic suboption of -qxf2003 272
 PORT @PROCESS directive 216
 POSITION @PROCESS directive 218
 POSIX pthreads
 API support 25
 runtime libraries 36
 POWER3, POWER4, POWER5, or PowerPC systems
 compiling programs for 31
 PowerPC systems
 compiling programs for 31
 precision of real data types 127, 223
 preprocessing Fortran source with the C preprocessor 31
 problem determination 291
 procedure trace 162, 163, 164, 165
 prof command 27
 profile-directed feedback (PDF) 208
 -qpdf1 compiler option 208
 -qpdf2 compiler option 208
 profiling 111
 -qpdf1 compiler option 208
 -qpdf2 compiler option 208
 profiling data files 27
 Program Editor 21
 promoting integer arguments to intrinsic procedures 269
 promotion of data types with -qautodbl option 311

ptevrpl suboption of -qalias 114

Q

QCOUNT @PROCESS directive 222
qdirectstorage compiler option 141
quiet NaN 174
quiet NaN suboption of -qflttrap 158

R

rand routine, naming restriction for 34
random runtime option 45
READ statements past end-of-file 269
README file 7
REAL data types 127
REALSIZE @PROCESS directive 223
RECUR @PROCESS directive 225
recursion 225, 229
register flushing 187
resetpdf command 210
return code
 from compiler 292
 from Fortran programs 292
rpm command 16
rrm suboption of -qfloat 250
run time
 exceptions 47
 options 36
running a program 35
running the compiler 21
runtime
 libraries 25
 problems 297
runtime environment
 external names in 309
runtime options 37

S

S error severity 291
SAA @PROCESS directive 228
SAA FORTRAN definition, checking
 conformance to 4
SAVE @PROCESS directive 229
scratch_vars runtime option 10, 45
setlocale libc routine 36
setrteopts service and utility
 procedure 36
severe error 291
sh shell 8
shared (dynamic) library 24
shared libraries 309
shared object files 25
shared objects 200
 -qmkshrobj 200
shared-memory parallelism (SMP) 237
showpdf 210
SIGN intrinsic, effect of
 -qxf90=signedzero on 270
signal handling 47
signedzero suboption of -qxf90 270
SIGTRAP signal 47
small suboption of -qtbtable 257
SMP
 programs, compiling 25

smplibraries attribute of configuration
 file 16
softeof suboption of -qxf77 269
SOURCE @PROCESS directive 242
source file options 29
source files 25
 allowing suffixes other than .f 16
 preserving path names for
 debugging 161
 specifying options in 29
source section in compiler listing 302
source-code conformance checking 4
source-level debugging support 6
space problems 294
SPILLSIZE @PROCESS directive 107,
 243
ssuffix attribute of configuration file 16
stack
 limit 294
stackprotect
 stackprotect 244
standards, language 3
static library 24
static linking 34
static storage, alignment of arrays in 118
std suboption of -qalias 114
storage limits 294
storage relationship between data
 objects 310
storage-associated arrays, performance
 implications of 114
STRICT @PROCESS directive 247
strictieemod @PROCESS directive 251
suffix, allowing other than .f on source
 files 16
suffixes for source files 252, 253
summary of command line options 77
SWAPOMP @PROCESS directive 256
symbolic debugger support 6
system problems 294

T

target machine, compiling for 120
temporary arrays, reducing 114
temporary file directory 10
text editors 21
threads, controlling 41
throughput for I/O, increasing with data
 striping 118
times routine, naming restriction for 34
TMPDIR environment variable 47, 297
 compile time 10
trace 162, 163, 164, 165
Trace/breakpoint trap 47
traceback listing 234, 298
transformation report section in compiler
 listing 303
transformations, control of 247
trigger_constant
 IBM* 140
 IBMT 259
 setting values 139
trigraphs 33
tuning 260
 -qarch compiler option 260
 -qtune compiler option 260

typeless constants and character
 constants 135
tapestmt and notyapestmt suboptions of
 -qport 216

U

U error severity 291
ufmt_littleendian runtime option 45
ulimit command 294
UNDEF @PROCESS directive 262, 282
underflow suboption of -qflttrap 159
unformatted data files, little-endian
 I/O 45
Unicode data 199
unit_vars runtime option 10, 46
UNIVERSAL setting for locale 199
unrecoverable error 291
unrolling DO LOOPS 262
UNWIND @PROCESS directive 264
use attribute of configuration file 14
UTF-8 encoding for Unicode data 199
uwidth runtime option 46

V

value relationships between data
 objects 310
vector processing 234
vi text editor 21

W

W error severity 291
warning error 291
what command 16
WRITE statements past end-of-file 269

X

XFLAG(OLDTAB) @PROCESS
 directive 268
xl_trbk library procedure 298
xl_trce exception handler 234
xlf attribute of configuration file 15
xlf_r command
 for compiling SMP programs 25
XLF_USR_CONFIG environment
 variable 47
xlf.cfg configuration file 14
xlf.cfg.nn configuration file 102
xlf.cfgconfiguration file 102
XLF2003 @PROCESS directive 272
XLF77 @PROCESS directive 268
XLF90 @PROCESS directive 270
xlf90_r command
 for compiling SMP programs 25
xlf95_r command
 for compiling SMP programs 25
xlfopt attribute of configuration file 15
XLFRTOPTS environment variable 36,
 37
XLFSCRATCH_unit environment
 variable 10, 45

XLFUNIT_unit environment variable 10,
46
XLINES @PROCESS 275
XOR 269
XREF @PROCESS directive 277
xrf_messages runtime option 46

Z

z suboption of -y 287
zero suboption of -qieee 170
zerodivide suboption of -qflttrap 159
zeros (leading), in output 269
ZEROSIZE @PROCESS directive 278



Program Number: 5724-X16

Printed in USA

SC23-8609-00

