XL Fortran Enterprise Edition for AIX

IBM

User's Guide

Version 9.1

SC09-7898-00

XL Fortran Enterprise Edition for AIX

User's Guide

Version 9.1

SC09-7898-00

Note!
Before using this information and the product it supports, be sure to read the general information under "Notices" on page 427.

First Edition (August 2004)

This edition applies to IBM XL Fortran Enterprise Edition (Program 5724-I08), Version 9.1, and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

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

- Internet: compinfo@ca.ibm.com

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

Contents

Figures

What's New for XL Fortran

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

New or changed compiler options and suboptions:
- The **-qflttrap=nanq** suboption detects all NaN values handled or generated by floating point instructions, including those not created by invalid operations.
- The **-qport=nullarg** suboption treats an empty argument, which is delimited by a left parenthesis and a comma, two commas, or a comma and a right parenthesis, as a null argument.
- The **-qmodule=mangle81** option provides compatibility with Version 8.1 module naming conventions for non-intrinsic modules.
- The **-qsaveopt** option saves the command-line options used for compiling a source file in the corresponding object file.
- The **-qversion** option provides the version and release for the invoking compiler.

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

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

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

Other features:

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

Introduction

This document describes Version 9.1 of IBM® XL Fortran Enterprise Edition and explains how to compile, link, and run programs that are written in the Fortran language.

How to Use This Document

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

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

This document is not the place to find help on:

Installation,
which is covered in the documents that are listed in "XL Fortran and Operating System Publications" on page 4.

Writing Fortran programs,
which is covered in the *XL Fortran Enterprise Edition for AIX Language Reference*.

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

The second part discusses more general software-development topics.

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

Set up the compiler for yourself or someone else,
read "Where to Find Installation Instructions" on page 11.

Upgrade from an earlier version of the XL Fortran compiler,
read "Avoiding or Fixing Upgrade Problems" on page 25.

Create customized compiler defaults,
read "Customizing the Configuration File" on page 15.

Understand what all the compiler options are for and how they relate to each other,
browse through "Summary of the XL Fortran Compiler Options" on page 67.

Look up a particular option by name,
scan alphabetically through "Detailed Descriptions of the XL Fortran Compiler Options" on page 90.

Port a program to XL Fortran,
read "Options for Compatibility" on page 79 to see what options you may need; then read "Porting Programs to XL Fortran" on page 397 for other porting information.

How to Read the Syntax Diagrams and Statements

This document uses notation often referred to as "railroad tracks" to illustrate syntax for Fortran statements and AIX commands. Syntax for compiler options is illustrated through statements using notation often referred to as "braces and brackets".

Fortran keywords are shown in uppercase: for example, **OPEN**, **COMMON**, and **END**. You must spell them exactly as they are shown, although they are not case-sensitive.

Variable names and user-specified names appear in lowercase italics; for example, *array_element_name*.

If a variable or user-specified name ends in *_list*, it means that you can provide a list of terms that are separated by commas.

You must enter punctuation marks, parentheses, arithmetic operators, and other special characters as part of the syntax.

Syntax Diagrams

- Read syntax diagrams from left to right and from top to bottom, following the path of the line:
  - The ►►─── symbol indicates the beginning of a command.
  - The ───► symbol indicates that the command syntax continues on the next line.
  - The ►─── symbol indicates that a command is continued from the previous line.
  - The ───►◄ symbol indicates the end of a command.
  - Diagrams of syntactical units smaller than complete statements start with the ►─── symbol and end with the ───► symbol.
  - Constructs, interface blocks and derived type definitions consist of several individual statements. For such items, individual syntax diagrams show the required order for the equivalent Fortran statements.
- Required items appear on the main path (the main path):

```
►►──command_name──required_argument──────────────────────────────►◄
```

- Optional items appear below the main path:

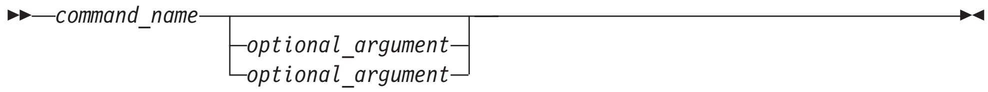
```
►►──command_name──┬───────────────────┬──────────────────────────►◄
                  └─optional_argument─┘
```

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

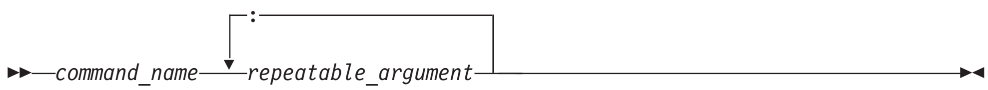
  If you *must* choose one of the items, one item of the stack appears on the main path:

```
►►──command_name──┬─required_argument─┬──────────────────────────►◄
                  └─required_argument─┘
```

  If choosing one of the items is optional, the entire stack appears below the main path:

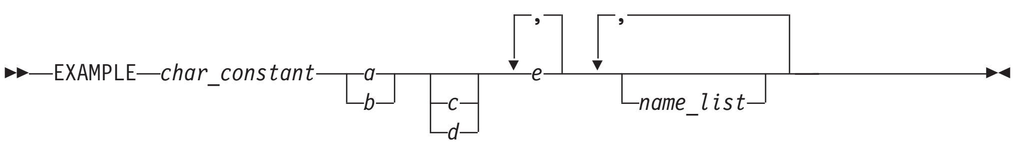


- An arrow returning to the left above the main line (a ″repeat arrow″) indicates an item that can be repeated and the separator character if it is other than a blank:



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

Example of a Syntax Diagram



Interpret the diagram as follows:

- Enter the keyword **EXAMPLE**.
- Enter a value for *char_constant*.
- Enter a value for *a* or *b*, but not for both.
- Optionally, enter a value for *c* or *d*.
- Enter at least one value for *e*. If you enter more than one value, you must put a comma between each.
- Optionally, enter the value of at least one *name* for *name_list*. If you enter more than one value, you must put a comma between each *name*.

Syntax Statements

Syntax statements are read from left to right:

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

Example of a Syntax Statement

EXAMPLE *char_constant* {*a*|*b*}[*c*|*d*]*e*[,*e*]... *name_list*{*name_list*}...

The following list explains the syntax statement:

- Enter the keyword **EXAMPLE**.
- Enter a value for *char_constant*.
- Enter a value for *a* or *b*, but not for both.
- Optionally, enter a value for *c* or *d*.
- Enter at least one value for *e*. If you enter more than one value, you must put a comma between each.

- Optionally, enter the value of at least one *name* for *name_list*. If you enter more than one value, you must put a comma between each *name*.

Note: The same example is used in both the syntax-statement and syntax-diagram representations.

Notes on the Examples in This Document

- The examples in this document are coded in a simple style that does not try to conserve storage, check for errors, achieve fast performance, or demonstrate all possible ways to do something.
- The examples in this document use the **xlf90**, **xlf90_r**, **xlf90_r7**, **xlf95**, **xlf95_r**, **xlf95_r7**, **xlf**, **xlf_r**, **xlf_r7**, **f77**, **fort77**, **f90**, and **f95** compiler invocation commands interchangeably. For more substantial source files, one of these commands may be more suitable than the others, as explained in "Compiling XL Fortran Programs" on page 29.
- Some sample programs from this document and some other programs that illustrate ideas presented in this document are in the directory **/usr/lpp/xlf/samples**.

Notes on the Terminology in This Document

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

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

Typographical Conventions

This document uses the following methods to differentiate text:

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

Related Documentation

You can refer to the following publications for additional information:

XL Fortran and Operating System Publications

- *IBM XL Fortran Enterprise Edition for AIX Language Reference* describes the XL Fortran programming language.
- The *AIX Installation Guide* covers all aspects of the standard AIX installation procedure. XL Fortran supplies brief installation instructions that explain how the general-purpose installation procedures apply to this licensed program.
- *AIX Commands Reference* (mutivolume set) contains extensive examples, as well as detailed descriptions of the AIX commands and their available flags. In particular, it describes the **ld** command (linker invocation).
- *AIX Performance Management Guide* explains how to maximize the performance of components of the AIX operating system.
- *AIX Technical Reference: Base Operating System and Extensions Volume 1* describes the Basic Linear Algebra Subroutines (BLAS), AIX subroutines, and AIX system calls.

- *General Programming Concepts: Writing and Debugging Programs* tells you how to write software that works properly in different countries and national languages.

Other Publications

These documents are also relevant to XL Fortran features:

- *Engineering and Scientific Subroutine Library Guide and Reference* gives information about the Engineering and Scientific Subroutine Library (ESSL) routines.
- *Parallel Engineering and Scientific Subroutine Library Guide and Reference* gives information about the Parallel Engineering and Scientific Subroutine Library (PESSL) routines.

Standards Documents

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

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

Overview of XL Fortran Features

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

Hardware and Operating-System Support

The XL Fortran Enterprise Edition Version 9.1 compiler is supported on the Version 5.1, or higher, AIX operating system. See the *XL Fortran Enterprise Edition for AIX Installation Guide* and README file for a list of requirements.

The compiler, its generated object programs, and run-time library will run on all RISC System/6000® (RS/6000®) or pSeries® systems with the required software, disk space, and virtual storage.

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

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

Language Support

The XL Fortran language consists of the following:

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

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

Migration Support

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

To protect your investment in FORTRAN 77 source code, you can easily invoke the compiler with a set of defaults that provide backward compatibility with earlier versions of XL Fortran. The **xlf**, **xlf_r**, **xlf_r7**, **f77**, and **fort77** commands provide maximum compatibility with existing FORTRAN 77 programs. The default options provided with the **xlf90**, **xlf90_r**, and **xlf90_r7**commands give access to the full range of Fortran 90 language features. The default options provided with the **xlf95**, **xlf95_r**, and **xlf95_r7** commands give access to the full range of Fortran 95 language features.

To protect your investments in FORTRAN 77 object code, you can link Fortran 90 and Fortran 95 programs with existing FORTRAN 77 object modules and libraries. See "Linking New Objects with Existing Ones" on page 45 for details.

Source-Code Conformance Checking

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

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

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

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

Highly Configurable Compiler

You can invoke the compiler by using the**xlf**, **xlf_r**, **xlf_r7**, **xlf90**, **xlf90_r**, **xlf90_r7**, **xlf95**, **xlf95_r**, **xlf95_r7**, **f77**, or **fort77** command. The **xlf**, **xlf_r**, **xlf_r7**, and **f77** commands maintain maximum compatibility with the behavior and I/O formats of XL Fortran Version 2. The **xlf90**, **xlf90_r**, and **xlf90_r7** commands provide more Fortran 90 conformance and some implementation choices for efficiency and usability. The **xlf95**, **xlf95_r**, and **xlf95_r7** commands provide more Fortran 95 conformance and some implementation choices for efficiency and usability. The **fort77** command provides maximum compatibility with the XPG4 behavior.

The main difference between the set of **xlf_r**, **xlf90_r**, **xlf90_r7**, **xlf95_r**, and **xlf95_r7** commands and the set of **xlf**, **xlf90**, **xlf95**, **f77**, and **fort77** commands is that the first set links and binds the object files to the thread-safe components (libraries, crt0_r.o, 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/xlf.cfg:xlf_r
```

You can control the actions of the compiler through a set of options. The different categories of options help you to debug, to optimize and tune program performance, to select extensions for compatibility with programs from other platforms, and to do other common tasks that would otherwise require changing the source code.

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

For information on:
- The configuration file, see "Customizing the Configuration File" on page 15
- The invocation commands, see "Compiling XL Fortran Programs" on page 29
- The compiler options, see "Summary of the XL Fortran Compiler Options" on page 67 and "Detailed Descriptions of the XL Fortran Compiler Options" on page 90
- Compiler return codes, see "Understanding XL Fortran Messages" on page 370

Diagnostic Listings

The compiler output listing has optional sections that you can include or omit. For information about the applicable compiler options and the listing itself, refer to "Options That Control Listings and Messages" on page 77 and "Understanding XL Fortran Compiler Listings" on page 389.

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

Symbolic Debugger Support

You can use **dbx**, the IBM Distributed Debugger (a technology preview version), and other symbolic debuggers for your programs.

Program Optimization

The XL Fortran compiler helps you control the optimization of your programs:
- You can select different levels of compiler optimizations.
- You can turn on separate optimizations for loops, floating point, and other categories.
- You can optimize a program for a particular class of machines or for a very specific machine configuration, depending on where the program will run.

"Optimizing XL Fortran Programs" on page 305 is a road map to these features.

Documentation and Online Help

XL Fortran provides product documentation in the following formats:
- Readme files
- Installable man pages
- A searchable, HTML-based help system
- Portable Document Format (PDF) documents

These items are located, or accessed as follows:

Readme files The readme files are located in **/usr/xlf/** and in the root directory of the installation CD.

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

HTML-based help system
An Information Center of searchable HTML files which can be installed on an intranet and accessed by pointing the browser to http: server_name:5312/help/index.jsp. The product help system is also viewable online at http://www.ibm.com/software/awdtools/fortran/xlfortran/library.

PDF documents
The PDF files are located in the **/usr/xlf/pdf** directory. They are viewable and printable from the Adobe Acrobat Reader. If you do not have the Adobe Acrobat Reader installed, you can download it from http://www.adobe.com.

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

For the latest information about XL Fortran Enterprise Edition, visit the product web sitessites:
- The Help Center at http://www.ibm.com/software/awdtools/fortran/xlfortran/library.
- The product support site at http://www.ibm.com/software/awdtools/fortran/xlfortran/support.

Setting Up and Customizing XL Fortran

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

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

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

Where to Find Installation Instructions

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

1. Read the file called **/usr/lpp/xlf/DOC/README.FIRST**, and follow any directions it gives. It contains information that you should know and possibly distribute to other people who use XL Fortran.
2. Read any *Installation Guide* document that comes with the compiler to see if there are any important notices you should be aware of or any updates you might need to apply to your system before doing the installation.
3. Read the *AIX Installation Guide* to understand the full procedure for installing optional software from the distribution medium. This document provides good step-by-step instructions that should answer all your questions and should help you to solve any installation problems.

   If you are already experienced with AIX software installation, you can use the **installp** or **smit installp** 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 under **/usr** contain XL Fortran components:

- **/usr/bin** contains the compiler invocation commands.
- **/usr/lib** contains the libraries.
- **/usr/lpp/xlf** contains executables and files that the compiler needs.
- **/usr/include** contains the include files, some of which contain definitions that XL Fortran uses.
- **/usr/lib/nls/msg** contains the message catalog files that XL Fortran uses.
- **/usr/lpp/xlfrtemsg** contains the default message catalog files that are used by XL Fortran programs.
- **/usr/share/man/cat1** contains the compiler man pages.
- **/usr/share/man/info/en_US/xlf/pdf** contains the PDF format of the English XL Fortran publications.
- **/usr/share/man/info/en_US/xlf/html** contains the HTML format of the English XL Fortran publications.

- **/usr/share/man/info/en_US/xlf/postscript** contains the PostScript format of the English XL Fortran publications.

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

Correct Settings for Environment Variables

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

Environment Variable Basics

You can set the environment variables from shell command lines or from within shell scripts. If you are not sure which shell is in use, a quick way to find out is to issue an **echo $0**. This provides a different result in each shell:

```
$ sh
$ echo $0
sh
$ ksh
$ echo $0
ksh
$ csh
% echo $0
No file for $0.
%
```

The Bourne shell path is **/bin/sh**, the Korn shell path is **/bin/ksh**, and the C shell path is **/bin/csh**.

To set the environment variables so that everyone on the system has access to them, set the variables in the file **/etc/profile** (for the Bourne or the Korn shell), or set the variables in the file **/etc/csh.login** or in the file **/etc/csh.cshrc** (for the C shell). To set them for a specific user only, add the appropriate commands to the appropriate **.profile** or **.cshrc** file in the user's home directory. The variables are set the next time the user logs on.

For more information about setting environment variables, see the *AIX Commands Reference*.

The following examples show how to set environment variables from various shells.

From the Bourne or Korn shell:

```
NLSPATH=/usr/lib/nls/msg/%L/%N:/usr/lib/nls/msg/prime/%N
LANG=en_US
TMPDIR=/home/joe/temp
export LANG NLSPATH TMPDIR
```

From the C shell:

```
setenv LANG en_US
setenv NLSPATH /usr/lib/nls/msg/%L/%N:/usr/lib/nls/msg/prime/%N
setenv TMPDIR /home/joe/temp
```

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 **ksh** notation instead of repeating the syntax for all shells.

Environment Variables for National Language Support

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

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

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

For example, to specify the Japanese locale with the **IBM_eucJP** code page, use the following commands from the Bourne or Korn shell:

```
LANG=ja_JP
NLSPATH=/usr/lib/nls/msg/%L/%N:/usr/lib/nls/msg/prime/%N
export LANG NLSPATH
```

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

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

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

For example:

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

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

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

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

Notes:

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

Related Information: "Selecting the Language for Run-Time Messages" on page 50.

See *General Programming Concepts: Writing and Debugging Programs* for more information about National Language Support environment variables and locale concepts.

LIBPATH:Setting Library Search Paths

Under normal circumstances, you only need **LIBPATH** if libraries are located in different directories at run time from those that they are in at compile time. To use **LIBPATH**, set it at run time to the names of any directories that contain required user libraries, plus **/usr/lib**:

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

# When the libraries are in the same directories as at compile
# time, the program finds them.
a.out

# If libmylib1.a and libmylib2.a are moved to /usr/lib/mydir3,
# you must set the LIBPATH variable:
export LIBPATH=/usr/lib/mydir3:/usr/lib
a.out
```

When running the compiler, ensure that the library **libxlf90.a** is in **/usr/lib** or is in a directory named in the **LIBPATH** setting. Otherwise, you cannot run the compiler, because it is dynamically linked with the **libxlf90.a** library.

PDFDIR: Specifying the Directory for PDF Profile Information

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

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

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

TMPDIR: Specifying a Directory for Temporary Files

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

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

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

XLFSCRATCH_unit: Specifying Names for Scratch Files

To give a specific name to a scratch file, you can set the run-time option **scratch_vars=yes**; then set one or more environment variables with names of the form **XLFSCRATCH_***unit* to file names to use when those units are opened as scratch files. See "Naming Scratch Files" on page 332 for examples.

XLFUNIT_unit: Specifying Names for Implicitly Connected Files

To give a specific name to an implicitly connected file or a file opened with no **FILE=** specifier, you can set the run-time option **unit_vars=yes**; then set one or more environment variables with names of the form **XLFUNIT_***unit* to file names. See "Naming Files That Are Connected with No Explicit Name" on page 331 for examples.

Customizing the Configuration File

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

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

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

```
ln -s /bin/xlf95 /home/lisa/bin/my_xlf95
```

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

Notes:

1. The configuration file contains other named stanzas to which you may want to link.
2. If you make any changes to the default configuration file and then move or copy your makefiles to another system, you will also need to copy the changed configuration file.

3. Installing a compiler program temporary fix (PTF) or an upgrade may overwrite the **/etc/xlf.cfg** file. Therefore, be sure to save a copy of any modifications you have made before doing such an installation.
4. If you upgrade the operating system, you must change the symbolic link in **/etc/xlf.cfg** to point to the correct version of the configuration file.
5. 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.
6. The **xlf_r**, **xlf90_r**, and **xlf95_r** stanzas support the operating system default thread interface: that is, the POSIX1003.1-1996 standard for AIX Version 5.1 and higher levels of the AIX operating system.
7. If you are mixing Message-Passing Interface (MPI) and threaded programming, use the appropriate stanza in the **xlf.cfg** file to link in the proper libraries and to set the correct default behavior:

   mpxlf Specifies **xlf** and **f77** behavior when using MPI.

   mpxlf_r Specifies **xlf_r** behavior when mixing MPI and threaded programming. You access this stanza by specifying the **mpxlf_r** command.

   mpxlf_r7 Specifies **xlf_r7** behavior when mixing MPI and threaded programming. You access this stanza by specifying the **mpxlf_r** command with the **-d7** option. The level of POSIX pthreads API support is Draft 7.

   mpxlf90 Specifies **xlf90** behavior when using MPI.

   mpxlf90_r Specifies **xlf90_r** behavior when mixing MPI and threaded programming. You access this stanza by specifying the **mpxlf90_r** command.

   mpxlf90_r7 Specifies **xlf90_r7** behavior when mixing MPI and threaded programming. You access this stanza by specifying the **mpxlf90_r** command with the **-d7** option. The level of POSIX pthreads API support is Draft 7.

   mpxlf95 Specifies **xlf95** behavior when using MPI.

   mpxlf95_r Specifies **xlf95_r** behavior when mixing MPI and threaded programming. You access this stanza by specifying the **mpxlf95_r** command.

   mpxlf95_r7 Specifies **xlf95_r7** behavior when mixing MPI and threaded programming. You access this stanza by specifying the **mpxlf95_r** command with the **-d7** option. The level of POSIX pthreads API support is Draft 7.

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.

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

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

code
The absolute path name of the optimizing code generator.

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

as
The absolute path name of the assembler.

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

ld
The absolute path name of the linker.

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

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

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

cppoptions
A string of options that are separated by commas, to be processed by **cpp** (the C preprocessor) as if you entered them on the command line before any other option. This attribute is needed because some **cpp** options are usually required to produce output

that can be compiled by XL Fortran. The default option is **-C**, which preserves any C-style comments in the output.

fsuffix
: The allowed suffix for Fortran source files. The default is **f**. The compiler requires that all source files in a single compilation have the same suffix. Therefore, to compile files with other suffixes, such as **f95**, you must change this attribute in the configuration file or use the **-qsuffix** compiler option. For more information on **-qsuffix**, see "-qsuffix Option" on page 244.

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.

proflibs
: **-L** options, which are separated by commas, that specify the path where the linker searches for additional libraries for profiled programs.

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

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

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

bolt
: Absolute path name of the binder.

defaultmsg
: Absolute path name of the default message files.

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

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

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

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

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

What a Configuration File Looks Like

The following is an example of a configuration file:

```
* Standard Fortran compiler
xlf95:    use        = DEFLT
          libraries  = -lxlf90,-L/usr/lpp/xlf/lib,-lxlopt,-lxlf,-lxlomp_ser,
                       -lm,-lc
          proflibs   = -L/lib/profiled,-L/usr/lib/profiled
          options    = -qfree=f90

* Alias for standard Fortran compiler
```

```
f95:      use        = DEFLT
          libraries  = -lxlf90,-L/usr/lpp/xlf/lib,-lxlopt,-lxlf,-lxlomp_ser,
                       -lm,-lc
          proflibs   = -L/lib/profiled,-L/usr/lib/profiled
          options    = -qfree=f90
          fsuffix    = f95

* Fortran 90 compiler
xlf90:    use        = DEFLT
          libraries  = -lxlf90,-L/usr/lpp/xlf/lib,-lxlopt,-lxlf,-lxlomp_ser,
                       -lm,-lc
          proflibs   = -L/lib/profiled,-L/usr/lib/profiled
          options    = -qxlf90=noautodealloc:nosignedzero,-qfree=f90

* Alias for Fortran 90 compiler
f90:      use        = DEFLT
          libraries  = -lxlf90,-L/usr/lpp/xlf/lib,-lxlopt,-lxlf,-lxlomp_ser,
                       -lm,-lc
          proflibs   = -L/lib/profiled,-L/usr/lib/profiled
          options    = -qxlf90=noautodealloc:nosignedzero,-qfree=f90
          fsuffix    = f90

* Original Fortran compiler
xlf:      use        = DEFLT
          libraries  = -lxlf90,-L/usr/lpp/xlf/lib,-lxlopt,-lxlf,-lxlomp_ser,
                       -lm,-lc
          proflibs   = -L/lib/profiled,-L/usr/lib/profiled
          options    = -qnozerosize,-qsave,-qalias=intptr,-qposition=appendold,
                       -qxlf90=noautodealloc:nosignedzero,-qxlf77=intarg:intxor:
                       persistent:noleadzero:gedit77:noblankpad:oldboz:softeof

* Alias for original Fortran compiler
f77:      use        = DEFLT
          libraries  = -lxlf90,-L/usr/lpp/xlf/lib,-lxlopt,-lxlf,-lxlomp_ser,
                       -lm,-lc
          proflibs   = -L/lib/profiled,-L/usr/lib/profiled
          options    = -qnozerosize,-qsave,-qalias=intptr,-qposition=appendold,
                       -qxlf90=noautodealloc:nosignedzero,-qxlf77=intarg:intxor:
                        persistent:noleadzero:gedit77:noblankpad:oldboz:softeof

* Alias for original Fortran compiler, used for XPG4 compliance
fort77:   use        = DEFLT
          libraries  = -lf,-L/usr/lpp/xlf/lib,-lxlopt,-lxlf,-lxlomp_ser,
                       -lm,-lc
          proflibs   = -L/lib/profiled,-L/usr/lib/profiled
          options    = -qnozerosize,-qsave,-qalias=intptr,-qposition=appendold,
                       -qxlf90=noautodealloc:nosignedzero,-qxlf77=intarg:intxor:
                        persistent:noleadzero:gedit77:noblankpad:oldboz:softeof

* xlf with links to thread-safe components
xlf_r:    use        = DEFLT
        crt          = /lib/crt0_r.o
        mcrt         = /lib/mcrt0_r.o
        gcrt         = /lib/gcrt0_r.o
        libraries    = -lxlf90,-L/usr/lpp/xlf/lib,-lxlopt,-lxlf,-lxlomp_ser,
                       -lpthreads,-lm,-lc
        smplibraries = -lxlf90,-L/usr/lpp/xlf/lib,-lxlopt,-lxlf,-lxlsmp,
                       -lpthreads,-lm,-lc
        proflibs     = -L/lib/profiled,-L/usr/lib/profiled
          options    = -qthreaded,-qnozerosize,-qsave,-qalias=intptr,
                       -qposition=appendold,-qxlf90=noautodealloc:nosignedzero,
                       -qxlf77=intarg:intxor:persistent:noleadzero:gedit77:
                        noblankpad:oldboz:softeof

* xlf90 with links to thread-safe components
xlf90_r:  use        = DEFLT
          crt        = /lib/crt0_r.o
```

```
          mcrt       = /lib/mcrt0_r.o
          gcrt       = /lib/gcrt0_r.o
          libraries  = -lxlf90,-L/usr/lpp/xlf/lib,-lxlopt,-lxlf,-lxlomp_ser,
                         -lpthreads,-lm,-lc
          smplibraries = -lxlf90,-L/usr/lpp/xlf/lib,-lxlopt,-lxlf,-lxlsmp,
                           -lpthreads,-lm,-lc
          proflibs   = -L/lib/profiled,-L/usr/lib/profiled
          options    = -qxlf90=noautodealloc:nosignedzero,-qfree=f90,-qthreaded

* xlf95 with links to thread-safe components
xlf95_r:  use        = DEFLT
          crt        = /lib/crt0_r.o
          mcrt       = /lib/mcrt0_r.o
          gcrt       = /lib/gcrt0_r.o
          libraries  = -lxlf90,-L/usr/lpp/xlf/lib,-lxlopt,-lxlf,-lxlomp_ser,
                         -lpthreads,-lm,-lc
          smplibraries = -lxlf90,-L/usr/lpp/xlf/lib,-lxlopt,-lxlf,-lxlsmp,
                           -lpthreads,-lm,-lc
          proflibs   = -L/lib/profiled,-L/usr/lib/profiled
          options    = -qfree=f90,-qthreaded

* xlf with links to thread-safe components (51 POSIX Draft 7 Threads)
xlf_r7: use        = DEFLT
        crt        = /lib/crt0_r.o
        mcrt       = /lib/mcrt0_r.o
        gcrt       = /lib/gcrt0_r.o
        libraries  = -lxlfpthrds_compat,-lxlf90,-L/usr/lpp/xlf/lib,-lxlopt,
                     -lxlf,-lxlomp_ser,-lpthreads_compat,-lpthreads,-lm,-lc
        proflibs   = -L/lib/profiled,-L/usr/lib/profiled
        smplibraries = -lxlfpthrds_compat,-lxlf90,-L/usr/lpp/xlf/lib,-lxlopt,
                       -lxlf,-lxlsmp,-lpthreads_compat,-lpthreads,-lm,-lc
        options    = -qthreaded,-qnozerosize,-qsave,-qalias=intptr,
                     -qposition=appendold,-qxlf90=noautodealloc:nosignedzero,
                     -qxlf77=intarg:intxor:persistent:noleadzero:gedit77:
                      noblankpad:oldboz:softeof
 include_32 = -I/usr/lpp/xlf/include_d7

* xlf90 with links to thread-safe components (51 POSIX Draft 7 Threads)
xlf90_r7: use        = DEFLT
          crt        = /lib/crt0_r.o
          mcrt       = /lib/mcrt0_r.o
          gcrt       = /lib/gcrt0_r.o
          libraries  = -lxlfpthrds_compat,-lxlf90,-L/usr/lpp/xlf/lib,-lxlopt,
                       -lxlf,-lxlomp_ser,-lpthreads_compat,-lpthreads,-lm,-lc
          proflibs   = -L/lib/profiled,-L/usr/lib/profiled
          smplibraries = -lxlfpthrds_compat,-lxlf90,-L/usr/lpp/xlf/lib,-lxlopt,
                         -lxlf,-lxlsmp,-lpthreads_compat,-lpthreads,-lm,-lc
          options    = -qxlf90=noautodealloc:nosignedzero,-qfree=f90,-qthreaded
   include_32 = -I/usr/lpp/xlf/include_d7

* xlf95 with links to thread-safe components (51 POSIX Draft 7 Threads)
xlf95_r7: use        = DEFLT
          crt        = /lib/crt0_r.o
          mcrt       = /lib/mcrt0_r.o
          gcrt       = /lib/gcrt0_r.o
          libraries   = -lxlfpthrds_compat,-lxlf90,-L/usr/lpp/xlf/lib,-lxlopt,
                         -lxlf,-lxlomp_ser,-lpthreads_compat,-lpthreads,-lm,-lc
          proflibs   = -L/lib/profiled,-L/usr/lib/profiled
          smplibraries = -lxlfpthrds_compat,-lxlf90,-L/usr/lpp/xlf/lib,-lxlopt,
                         -lxlf,-lxlsmp,-lpthreads_compat,-lpthreads,-lm,-lc
          options    = -qfree=f90,-qthreaded
   include_32 = -I/usr/lpp/xlf/include_d7

* PE Fortran, with Fortran 95 behavior
mpxlf95:  use        = DEFLT
          libraries  = -L/usr/lpp/ppe.poe/lib,-L/usr/lpp/ppe.poe/lib/ip,-lmpi,
                         -lvtd,-lxlf90,-L/usr/lpp/xlf/lib,-lxlopt,-lxlf,
```

```
                     -lxlomp_ser,-lm,-lc
          proflibs   = -L/usr/lpp/ppe.poe/lib/profiled,-L/lib/profiled,
                       -L/usr/lib/profiled
          options    = -qfree=f90,-binitfini:poe_remote_main
          include    = -I/usr/lpp/ppe.poe/include

* PE Fortran, with Fortran 90 behavior
mpxlf90:  use        = DEFLT
          libraries  = -L/usr/lpp/ppe.poe/lib,-L/usr/lpp/ppe.poe/lib/ip,-lmpi,
                       -lvtd,-lxlf90,-L/usr/lpp/xlf/lib,-lxlopt,-lxlf,
                       -lxlomp_ser,-lm,-lc
          proflibs   = -L/usr/lpp/ppe.poe/lib/profiled,-L/lib/profiled,
                       -L/usr/lib/profiled
          options    = -qxlf90=noautodealloc:nosignedzero,-qfree=f90,-binitfini:
                        poe_remote_main
          include    = -I/usr/lpp/ppe.poe/include

* PE Fortran, with FORTRAN 77 behavior
mpxlf:    use        = DEFLT
          libraries  = -L/usr/lpp/ppe.poe/lib,-L/usr/lpp/ppe.poe/lib/ip,-lmpi,
                       -lvtd,-lxlf90,-L/usr/lpp/xlf/lib,-lxlopt,-lxlf,
                       -lxlomp_ser,-lm,-lc
          proflibs   = -L/usr/lpp/ppe.poe/lib/profiled,-L/lib/profiled,
                       -L/usr/lib/profiled
          options    = -qnozerosize,-qsave,-qalias=intptr,-qposition=appendold,
                       -qxlf90=noautodealloc:nosignedzero,-qxlf77=intarg:intxor:
                        persistent:noleadzero:gedit77:noblankpad:oldboz:softeof,
                        -binitfini:poe_remote_main
          include    = -I/usr/lpp/ppe.poe/include

* PE Fortran, with Fortran 95 behavior, and links to thread-safe components
mpxlf95_r:  use        = DEFLT
            crt        = /lib/crt0_r.o
            mcrt       = /lib/mcrt0_r.o
            gcrt       = /lib/gcrt0_r.o
            libraries  = -L/usr/lpp/ppe.poe/lib/threads,-L/usr/lpp/ppe.poe/
                         lib,-L/usr/lpp/ppe.poe/lib/ip,-L/lib/threads,-lmpi_r,
                         -lvtd_r,-lxlf90,-L/usr/lpp/xlf/lib,-lxlopt,-lxlf,
                         -lxlomp_ser,-lpthreads,-lm_r,-lm,-lc_r,-lc,
                         /usr/lpp/ppe.poe/lib/libc.a
            proflibs   = -L/usr/lpp/ppe.poe/lib/profiled/threads,
                         -L/usr/lpp/ppe.poe/lib/profiled,-L/lib/profiled,
                         -L/usr/lib/profiled
            smplibraries = -L/usr/lpp/ppe.poe/lib/threads,-L/usr/lpp/ppe.poe/
                           lib,-L/usr/lpp/ppe.poe/lib/ip,-L/lib/threads,-lmpi_r,
                           -lvtd_r,-lxlf90,-L/usr/lpp/xlf/lib,-lxlopt,-lxlf,
                           -lxlsmp,-lpthreads,-lm_r,-lm,-lc_r,-lc,
                           /usr/lpp/ppe.poe/lib/libc.a
            options    = -qthreaded,-qfree=f90,-binitfini:poe_remote_main
            include    = -I/usr/lpp/ppe.poe/include

* PE Fortran, with Fortran 90 behavior, and links to thread-safe components
mpxlf90_r:  use        = DEFLT
            crt        = /lib/crt0_r.o
            mcrt       = /lib/mcrt0_r.o
            gcrt       = /lib/gcrt0_r.o
            libraries  = -L/usr/lpp/ppe.poe/lib/threads,-L/usr/lpp/ppe.poe/lib,
                         -L/usr/lpp/ppe.poe/lib/ip,-L/lib/threads,-lmpi_r,
                         -lvtd_r,-lxlf90,-L/usr/lpp/xlf/lib,-lxlopt,-lxlf,
                         -lxlomp_ser,-lpthreads,-lm_r,-lm,-lc_r,-lc,
                         /usr/lpp/ppe.poe/lib/libc.a
            proflibs   = -L/usr/lpp/ppe.poe/lib/profiled/threads,
                         -L/usr/lpp/ppe.poe/lib/profiled,-L/lib/profiled,
                         -L/usr/lib/profiled
            smplibraries = -L/usr/lpp/ppe.poe/lib/threads,
                           -L/usr/lpp/ppe.poe/lib,-L/usr/lpp/ppe.poe/lib/ip,
                           -L/lib/threads,-lmpi_r,-lvtd_r,-lxlf90,
```

```
                      -L/usr/lpp/xlf/lib,-lxlopt,-lxlf,-lxlsmp,-lpthreads,
                      -lm_r,-lm,-lc_r,-lc,/usr/lpp/ppe.poe/lib/libc.a
            options    = -qxlf90=noautodealloc:nosignedzero,-qthreaded,
                      -qfree=f90,-binitfini:poe_remote_main
            include    = -I/usr/lpp/ppe.poe/include

* PE Fortran, with FORTRAN 77 behavior, and links to thread-safe components
mpxlf_r:  use        = DEFLT
          crt        = /lib/crt0_r.o
          mcrt       = /lib/mcrt0_r.o
          gcrt       = /lib/gcrt0_r.o
          libraries  = -L/usr/lpp/ppe.poe/lib/threads,-L/usr/lpp/ppe.poe/lib,
                       -L/usr/lpp/ppe.poe/lib/ip,-L/lib/threads,-lmpi_r,-lvtd_r,
                       -lxlf90,-L/usr/lpp/xlf/lib,-lxlopt,-lxlf,-lxlomp_ser,
                       -lpthreads,-lm_r,-lm,-lc_r,-lc,
                        /usr/lpp/ppe.poe/lib/libc.a
          proflibs   = -L/usr/lpp/ppe.poe/lib/profiled/threads,
                       -L/usr/lpp/ppe.poe/lib/profiled,-L/lib/profiled,
                       -L/usr/lib/profiled
          smplibraries = -L/usr/lpp/ppe.poe/lib/threads,-L/usr/lpp/ppe.poe/lib,
                         -L/usr/lpp/ppe.poe/lib/ip,-L/lib/threads,-lmpi_r,
                         -lvtd_r,-lxlf90,-L/usr/lpp/xlf/lib,-lxlopt,-lxlf,
                         -lxlsmp,-lpthreads,-lm_r,-lm,-lc_r,-lc,
                         /usr/lpp/ppe.poe/lib/libc.a
          options    = -qthreaded,-qnozerosize,-qsave,-qalias=intptr,
                       -qposition=appendold,-qxlf90=noautodealloc:nosignedzero,
                       -qxlf77=intarg:intxor:persistent:noleadzero:gedit77:
                        noblankpad:oldboz:softeof,-binitfini:poe_remote_main
          include    = -I/usr/lpp/ppe.poe/include

* mpxlf95_r, links to thread-safe components (51 POSIX Draft 7 Threads)
mpxlf95_r7: use        = DEFLT
            crt        = /lib/crt0_r.o
            mcrt       = /lib/mcrt0_r.o
            gcrt       = /lib/gcrt0_r.o
            libraries  = -L/usr/lpp/ppe.poe/lib/threads,-L/usr/lpp/ppe.poe/lib,
                         -L/usr/lpp/ppe.poe/lib/ip,-L/lib/threads,-lmpi_r,
                         -lvtd_r,-lxlfpthrds_compat,-lxlf90,
                         -L/usr/lpp/xlf/lib,-lxlopt,-lxlf,-lxlomp_ser,
                         -lpthreads_compat,-lpthreads,-lm_r,-lm,-lc_r,-lc,
                         /usr/lpp/ppe.poe/lib/libc.a
            proflibs   = -L/usr/lpp/ppe.poe/lib/profiled/threads,
                         -L/usr/lpp/ppe.poe/lib/profiled,-L/lib/profiled,
                         -L/usr/lib/profiled
            smplibraries = -L/usr/lpp/ppe.poe/lib/threads,
                           -L/usr/lpp/ppe.poe/lib,-L/usr/lpp/ppe.poe/lib/ip,
                           -L/lib/threads,-lmpi_r,-lvtd_r,-lxlfpthrds_compat,
                           -lxlf90,-L/usr/lpp/xlf/lib,-lxlopt,-lxlf,-lxlsmp,
                           -lpthreads_compat,-lpthreads,-lm_r,-lm,-lc_r,-lc,
                           /usr/lpp/ppe.poe/lib/libc.a
            options    = -qthreaded,-qfree=f90,-binitfini:poe_remote_main
            include    = -I/usr/lpp/ppe.poe/include
     include_32 = -I/usr/lpp/xlf/include_d7

* mpxlf90_r, links to thread-safe components (51 POSIX Draft 7 Threads)
mpxlf90_r7: use        = DEFLT
            crt        = /lib/crt0_r.o
            mcrt       = /lib/mcrt0_r.o
            gcrt       = /lib/gcrt0_r.o
            libraries  = -L/usr/lpp/ppe.poe/lib/threads,-L/usr/lpp/ppe.poe/lib,
                         -L/usr/lpp/ppe.poe/lib/ip,-L/lib/threads,-lmpi_r,
                         -lvtd_r,-lxlfpthrds_compat,-lxlf90,-L/usr/lpp/xlf/lib,
                         -lxlopt,-lxlf,-lxlomp_ser,-lpthreads_compat,-lpthreads,
                         -lm_r,-lm,-lc_r,-lc,/usr/lpp/ppe.poe/lib/libc.a
            proflibs   = -L/usr/lpp/ppe.poe/lib/profiled/threads,
                         -L/usr/lpp/ppe.poe/lib/profiled,-L/lib/profiled,
                         -L/usr/lib/profiled
```

```
          smplibraries = -L/usr/lpp/ppe.poe/lib/threads,
                         -L/usr/lpp/ppe.poe/lib,-L/usr/lpp/ppe.poe/lib/ip,
                         -L/lib/threads,-lmpi_r,-lvtd_r,-lxlfpthrds_compat,
                         -lxlf90,-L/usr/lpp/xlf/lib,-lxlopt,-lxlf,-lxlsmp,
                         -lpthreads_compat,-lpthreads,-lm_r,-lm,-lc_r,-lc,
                         /usr/lpp/ppe.poe/lib/libc.a
          options    = -qxlf90=noautodealloc:nosignedzero,-qthreaded,
                       -qfree=f90,-binitfini:poe_remote_main
          include    = -I/usr/lpp/ppe.poe/include
   include_32 = -I/usr/lpp/xlf/include_d7

* mpxlf_r, links to thread-safe components (51 POSIX Draft 7 Threads)
mpxlf_r7: use        = DEFLT
          crt        = /lib/crt0_r.o
          mcrt       = /lib/mcrt0_r.o
          gcrt       = /lib/gcrt0_r.o
          libraries  = -L/usr/lpp/ppe.poe/lib/threads,-L/usr/lpp/ppe.poe/lib,
                       -L/usr/lpp/ppe.poe/lib/ip,-L/lib/threads,-lmpi_r,-lvtd_r,
                       -lxlfpthrds_compat,-lxlf90,-L/usr/lpp/xlf/lib,-lxlopt,
                       -lxlf,-lxlomp_ser,-lpthreads_compat,-lpthreads,-lm_r,-lm,
                       -lc_r,-lc,/usr/lpp/ppe.poe/lib/libc.a
          proflibs   = -L/usr/lpp/ppe.poe/lib/profiled/threads,
                       -L/usr/lpp/ppe.poe/lib/profiled,-L/lib/profiled,
                       -L/usr/lib/profiled
          smplibraries = -L/usr/lpp/ppe.poe/lib/threads,-L/usr/lpp/ppe.poe/lib,
                         -L/usr/lpp/ppe.poe/lib/ip,-L/lib/threads,-lmpi_r,
                         -lvtd_r,-lxlfpthrds_compat,-lxlf90,-L/usr/lpp/xlf/lib,
                         -lxlopt,-lxlf,-lxlsmp,-lpthreads_compat,-lpthreads,
                         -lm_r,-lm,-lc_r,-lc,/usr/lpp/ppe.poe/lib/libc.a
          options    = -qthreaded,-qnozerosize,-qsave,-qalias=intptr,
                       -qposition=appendold,-qxlf90=noautodealloc:nosignedzero,
                       -qxlf77=intarg:intxor:persistent:noleadzero:gedit77:
                        noblankpad:oldboz:softeof,-binitfini:poe_remote_main
          include    = -I/usr/lpp/ppe.poe/include
   include_32 = -I/usr/lpp/xlf/include_d7

* Common definitions
DEFLT:    xlf        = /usr/lpp/xlf/bin/xlfentry
          crt        = /lib/crt0.o
          mcrt       = /lib/mcrt0.o
          gcrt       = /lib/gcrt0.o
          crt_64     = /lib/crt0_64.o
          mcrt_64    = /lib/mcrt0_64.o
          gcrt_64    = /lib/gcrt0_64.o
          include_32 = -I/usr/lpp/xlf/include
          include_64 = -I/usr/lpp/xlf/include
          fppv       = /usr/lpp/xlf/bin/fppv
          fppk       = /usr/lpp/xlf/bin/fppk
          dis        = /usr/lpp/xlf/bin/dis
          code       = /usr/lpp/xlf/bin/xlfcode
          hot        = /usr/lpp/xlf/bin/xlfhot
          ipa        = /usr/lpp/xlf/bin/ipa
          bolt       = /usr/lpp/xlf/bin/bolt
          defaultmsg = /usr/lpp/xlf/bin/default_msg
          as         = /bin/as
          ld         = /bin/ld
          cppoptions = -C
          options    = -bh:4,-bpT:0x10000000,-bpD:0x20000000
          oslevel    = 5.1
```

XL Fortran provides the library **libxlf90_r.a** in addition to **libxlf90_t.a**. The library **libxlf90_r.a** is a superset of **libxlf90_t.a**, which is a partial thread-support run-time library. The file **xlf.cfg** has been set up to link to **libxlf90_r.a** automatically when you use the **xlf90_r**, **xlf90_r7**, **xlf95_r**, **xlf95_r7**, **xlf_r**, and **xlf_r7** commands.

Related Information: You can use the "-F Option" on page 107 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:

```
lslpp -h "*xlf*"
```

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

To check the level of the compiler executable itself, issue the command:

```
what /usr/lpp/xlf/bin/xlfentry
```

If the compiler is installed in a different directory, use the appropriate path name for the **xlfentry** file.

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

Upgrading to XL Fortran Version 9

Here is some advice to help make the transition from an earlier version of the XL Fortran compiler as fast and simple as possible.

Things to Note in XL Fortran Version 9

Because XL Fortran Version 9.1 is highly compatible with XL Fortran Versions 8 through 3 inclusive, most of the advice in this section applies to upgrades from Version 2, or earlier levels of XL Fortran.

- The **xlf90**, **xlf90_r**, and **xlf90_r7** commands provide Fortran 90 conformance, and the **xlf95**, **xlf95_r**, and **xlf95_r7** commands provide Fortran 95 conformance. However, these commands may cause some problems with existing FORTRAN 77 programs. The **xlf**, **xlf_r**, **xlf_r7**, **f77**, and **fort77** commands avoid some of these problems by keeping the old behavior wherever possible.
- Fortran 90 introduced the idea of kind parameters for types. Except for the types complex and character, XL Fortran uses numeric kind parameters that correspond to the lengths of the types. For the type complex, the kind parameter is equal to the length of the real portion, which is half of the overall length. For the type character, the kind parameter is equal to the number of bytes that are required to represent each character, and this value is **1**. A FORTRAN 77 declaration that is written using the * extension for length specifiers can now be rewritten with a kind parameter:

  ```
  INTEGER*4  X   ! F77 notation with extension.
  INTEGER(4) X   ! F90 standard notation.
  COMPLEX*8  Y   ! *n becomes (n) for all types except
  COMPLEX(4) Y   ! COMPLEX, where the value is halved.
  ```

  This new form is the one we use consistently throughout the XL Fortran manuals.

Because the values of kind parameters may be different for different compilers, you may want to use named constants, placed in an include file or a module, to represent the kind parameters used in your programs. The **SELECTED_INT_KIND** and **SELECTED_REAL_KIND** intrinsic functions also let you determine kind values in a portable way.

- Fortran 90 introduced a standardized free source form for source code, which is different from the XL Fortran Version 2 free source form. The **-qfree** and **-k** options now use the Fortran 90 free source form; the Version 2 free source form is available through the option **-qfree=ibm**.
- The libraries that provide Fortran 90 and Fortran 95 support are **libxlf90_r.a** and **libxlf90.a**, located in **/usr/lib**. A **libxlf.a** library of stub routines is provided in **/usr/lib**, but it is only used for linking existing Version 1 or 2 object files or running existing executables. When a Version 1 or Version 2 object file calls entry points in **libxlf.a**, those entry points then call equivalent entry points in **libxlf90.a** (for single-threaded programs) or **libxlf90_r.a** (for multi-threaded programs). If you recompile such object files, the result could be improved I/O performance, because the entry points in **libxlf90.a** or **libxlf90_r.a** are called directly.

  Fortran provides the library **libxlf90_r.a**, in addition to **libxlf90_t.a**. The library **libxlf90_r.a** is a superset of **libxlf90_t.a**, which is a partial thread-support run-time library.

  The file **xlf.cfg** has been set up to link to **libxlf90_r.a** automatically when you use the **xlf90_r**, **xlf90_r7**, **xlf95_r**, **xlf95_r7**, **xlf_r**, or **xlf_r7** command. A single, combined thread-safe library, **libxlf90_r.a**, is provided to support both single- and multiple-threaded applications. The **libxlf90.a** library is a symbolic link to **libxlf90_r.a**.

Avoiding or Fixing Upgrade Problems

Although XL Fortran is generally backward-compatible with FORTRAN 77 programs, there are some changes in XL Fortran and the Fortran 90 and Fortran 95 languages that you should be aware of.

To preserve the behavior of existing compilation environments, the **xlf**, and **f77** commands both work as they did in earlier XL Fortran versions wherever possible. As you write entirely new Fortran 90 or Fortran 95 programs or adapt old programs to avoid potential problems, you can begin using the **xlf90** and **xlf95** commands, which use Fortran 90 and Fortran 95 conventions for source-code format.

Note that in the following table, you can substitute **xlf_r** or **xlf_r7** for **xlf**, **xlf90_r** or **xlf90_r7** for **xlf90**, and **xlf95_r** or **xlf95_r7** for **xlf95**.

Table 1. Potential Problems Migrating Programs to XL Fortran Version 9. The column on the right shows which problems you can avoid by using the **xlf** or **f77** command.

Potential Problem	Solution or Workaround	xlf Avoids?
	Compilation Problems	
New intrinsic procedure names may conflict with external procedure names. The intrinsic procedure is called instead of the external procedure.	Use the **-qextern** option, or insert **EXTERNAL** statements to avoid the ambiguity. Consider switching to the Fortran 90 or Fortran 95 procedure if it does what you want.	

Table 1. Potential Problems Migrating Programs to XL Fortran Version 9 (continued). The column on the right shows which problems you can avoid by using the **xlf** or **f77** command.

Potential Problem	**Solution or Workaround**	**xlf Avoids?**
The **.XOR.** intrinsic is not recognized.	Use the option **-qxlf77=intxor**.	✓
Zero-sized objects are not allowed by the compiler.	Use the **xlf90** or **xlf95** command, or use the **-qzerosize** option with the **xlf** or **f77** command.	
Performance / Optimization Problems		
Existing programs or programs linked with older XL Fortran object files run more slowly or do not show expected performance improvements on new hardware.	Recompile everything.	
Programs compiled with **-O3** or **-qhot** optimization behave differently from those unoptimized (different results, exceptions, or compilation messages).	Try adding the **-qstrict** option.	
The option combination **-O** and **-1** cannot be abbreviated **-O1**, to avoid misunderstandings. (There are **-O2**, **-O3**, **-O4**, and **-O5** optimization levels, but there is no **-O1**.)	Specify **-O** and **-1** as separate options.	
Programs that use integer **POINTER**s produce incorrect results when optimized.	Specify the option **-qalias=intptr** with the **xlf90** or **xlf95** command, or use the **xlf** command.	✓

Table 1. Potential Problems Migrating Programs to XL Fortran Version 9 (continued). The column on the right shows which problems you can avoid by using the **xlf** or **f77** command.

Potential Problem	Solution or Workaround	xlf Avoids?
Run-Time Problems		
Programs that read to the end of the file and then try to append records without first executing a **BACKSPACE** statement do not work correctly. The write requests generate error messages.	To compile existing programs, specify the option **-qxlf77=softeof** with the **xlf90** or **xlf95** command, or use the **xlf** command. For new programs, add the **BACKSPACE** statement before writing past the endfile record.	✓
Uninitialized variables are not necessarily set to zero, and programs that ran before may exceed the user stack limit. The reason is that the default storage class is now **AUTOMATIC**, rather than **STATIC** (an implementation choice allowed by the language).	Ensure that you explicitly initialize your variables, use the **-qsave** option with the **xlf90** or **xlf95** command, or add **SAVE** statements where needed in the source.	✓
Writing data to some files opened without a **POSITION=** specifier overwrites the files, instead of appending the data.	Use the option **-qposition=appendold**, or add POSITION= specifiers where needed.	✓
Newly compiled programs are unable to read existing data files containing **NAMELIST** data. The reason is that the Fortran 90 and Fortran 95 standards define a namelist format that is different from that used on AIX in the past.	Set the environment variable **XLFRTEOPTS** to the string **namelist=old**. The programs that produced the old **NAMELIST** data must be recompiled.	
Some I/O statements and edit descriptors accept or produce slightly different input and output. For example, real output now has a leading zero when appropriate. The changes to I/O formats are intended to be more usable and typical of industry practice, so you should try to use the defaults for any new data you produce.	When you need to maintain compatibility with existing data files, compile with the **xlf** command. If the incompatibility is due to a single specific I/O change, see if the **-qxlf77** option has a suboption for backward compatibility. If so, you can switch to the **xlf90** or **xlf95** command and use the **-qxlf77** option on programs that use the old data files.	✓
Numeric results and I/O output are not always exactly identical with XL Fortran Version 2. Certain implementation details of I/O, such as spacing in list-directed output and the meanings of some **IOSTAT** values, have changed since XL Fortran Version 2. (This entry is similar to the previous one except that these differences have no backward-compatibility switches.)	You may need to generate existing data files again or to change any programs that depend on these details. When no backward-compatibility switch is provided by the **-qxlf77** compiler option or **XLFRTEOPTS** run-time options, there is no way to get the old behavior back.	

Table 1. Potential Problems Migrating Programs to XL Fortran Version 9 (continued). The column on the right shows which problems you can avoid by using the **xlf** or **f77** command.

Potential Problem	Solution or Workaround	xlf Avoids?
SIGN(A,B) now returns **-|A|** when **B=-0.0**. Prior to XL Fortran Version 7.1, it returned **|A|**.	This behavior conforms with the Fortran 95 standard and is consistent with the IEEE standard for binary floating-point arithmetic. It occurs because the **-qxlf90=signedzero** option is turned on. Turn it off, or specify a command that does not use this option by default.	✓
A minus sign is printed for a negative zero in formatted output. A minus sign is printed for negative values that have an outputted form of zero (that is, in the case where trailing non-zero digits are truncated from the output so that the resulting output looks like zero). Prior to XL Fortran Version 7.1, minus signs were not printed in these situations.	This behavior conforms with the Fortran 95 standard and occurs because the **-qxlf90=signedzero** option is turned on. Turn it off, or specify a command that does not use this option by default.	✓

Related Information:

- "Compiling for Specific Architectures" on page 39
- "Setting Run-Time Options" on page 51
- "-qalias Option" on page 122
- "-qextern Option" on page 157
- "-qposition Option" on page 219
- "-qsave Option" on page 228
- "-qxlf77 Option" on page 261

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 Enterprise Edition for AIX Installation Guide* for details.

Editing, Compiling, Linking, and Running XL Fortran Programs

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

Prerequisite Information:
1. Before you can use the compiler, all the required AIX settings (for example, certain environment variables and storage limits) must be correct for your user ID; for details, see "Correct Settings for Environment Variables" on page 12.
2. Before using the compiler for a specialized purpose, such as porting or performance tuning, look at the categories of options in "Summary of the XL Fortran Compiler Options" on page 67 to see if XL Fortran already provides a solution.
3. To learn more about writing Fortran programs, refer to the *XL Fortran Enterprise Edition for AIX Language Reference*.

Note that you can substitute references to **libxlf90_r.a** in this section with references to **libxlf90.a**. This is because a link is provided from the **libxlf90.a** library to the **libxlf90_r.a** library. You do not need to manually link with separate libraries depending on whether you are creating a threaded or a non-threaded application. XL Fortran determines at run time whether your application is threaded.

Editing XL Fortran Source Files

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

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

Compiling XL Fortran Programs

To compile a source program, use one of the **xlf90**, **xlf90_r**, **xlf90_r7**, **xlf95**, **xlf95_r**, **xlf95_r7**, **xlf**, **xlf_r**, **xlf_r7**, **f77**, **fort77**, **f90**, or **f95** commands, which have the form:

```
                    ┌──────────────────────────────┐
                    ▼                              │
►►──┬─xlf90────┬────┬──────────────┬─input_file─┴──────────────────────►◄
    ├─xlf90_r──┤    └─cmd_line_opt─┘
    ├─xlf90_r7─┤
    ├─xlf95────┤
    ├─xlf95_r──┤
    ├─xlf95_r7─┤
    ├─xlf──────┤
    ├─xlf_r────┤
    ├─xlf_r7───┤
    ├─f77──────┤
    ├─fort77───┤
    ├─f90──────┤
    └─f95──────┘
```

These commands all accept essentially the same Fortran language. The main difference is that they use different default options (which you can see by reading the file **/etc/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**, **xlf_r7**, **xlf90_r**, **xlf90_r7**, **xlf95_r**, and **xlf95_r7** commands use the thread-safe components (libraries, **crt0_r.o**, and so on) to link and bind object files.

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

Table 2. XL Fortran Invocation Commands

Driver Invocation	Path or Location	Chief Functionality	Linked Libraries
xlf90	`/usr/bin`	Fortran 90	libxlf90.a
xlf90_r	`/usr/bin`	Threadsafe Fortran 90, operating system default POSIX pthreads API	libxlf90_r.a
xlf90_r7	`/usr/bin`	Threadsafe Fortran 90, Draft 7 POSIX pthreads API	libxlf90_r.a
xlf95	`/usr/bin`	Fortran 95	libxlf90.a
xlf95_r	`/usr/bin`	Threadsafe Fortran 95, operating system default POSIX pthreads API	libxlf90_r.a
xlf95_r7	`/usr/bin`	Threadsafe Fortran 95, Draft 7 POSIX pthreads API	libxlf90_r.a
xlf	`/usr/bin`	FORTRAN 77	libxlf90.a
xlf_r	`/usr/bin`	Threadsafe FORTRAN 77, operating system default POSIX pthreads API	libxlf90_r.a
xlf_r7	`/usr/bin`	Threadsafe FORTRAN 77, Draft 7 POSIX pthreads API	libxlf90_r.a
f77 or **fort77**	`/usr/bin`	FORTRAN 77	libxlf90.a
f90	`/usr/bin`	Fortran 90	libxlf90.a
f95	`/usr/bin`	Fortran 95	libxlf90.a

The invocation commands have the following implications for directive triggers:

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

If you specify the **-qsmp** compiler option, the following occurs:
- The compiler turns on automatic parallelization.
- The compiler recognizes the **IBMP**, **IBMT**, **IBM***, **SMP$**, and **$OMP** directive triggers.

XL Fortran provides the library **libxlf90_r.a**, in addition to **libxlf90_t.a**. The library **libxlf90_r.a** is a superset of **libxlf90_t.a**. The file **xlf.cfg** is set up to link to **libxlf90_r.a** automatically when you use the **xlf90_r**, **xlf90_r7**, **xlf95_r**, **xlf95_r7**, **xlf_r**, and **xlf_r7** commands.

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

When you bind a multi-threaded executable with multiple Fortran threads, to link in routines in **libxlf90_r.a**, **-lxlf90_r** should appear instead of either **-lxlf90_t** or **-lxlf90** in the command line. Note that using the **xlf_r**, **xlf_r7**, **xlf90_r**, **xlf90_r7**, **xlf95_r**, or **xlf95_r7** invocation command ensures the proper linking.

Compiling XL Fortran Version 2 Programs

xlf maintains, wherever possible, compatibility with existing programs by using the same I/O formats as earlier versions of XL Fortran and some implementation behavior compatible with FORTRAN 77.

f77 is identical to **xlf** (assuming that the configuration file has not been customized).

You may find that you need to continue using these commands for compatibility with existing makefiles and build environments. However, be aware that programs that you compile with these commands may not conform to the Fortran 90 or Fortran 95 standard in subtle ways.

Compiling Fortran 90 or Fortran 95 Programs

The **f90**, **xlf90**, **xlf90_r**, and **xlf90_r7** commands make your programs conform more closely to the Fortran 90 standard than do the **xlf**, **xlf_r**, **xlf_r7**, and **f77**/**fort77** commands. The **f95**, **xlf95**, **xlf95_r**, and **xlf95_r7** commands make your programs conform more closely to the Fortran 95 standard than do the **xlf**, **xlf_r**, **xlf_r7**, and **f77**/**fort77** commands. **f90**, **xlf90**, **xlf90_r**, **xlf90_r7**, **f95**, **xlf95**, **xlf95_r**, and **xlf95_r7** are the preferred commands for compiling any new programs. They all accept Fortran 90 free source form by default; to use them for fixed source form, you must use the **-qfixed** option. I/O formats are slightly different between these commands and the other commands. I/O formats also differ between the set of

xlf90, **xlf90_r**, and **xlf90_r7** commands and the set of **xlf95**, **xlf95_r**, and **xlf95_r7** commands. We recommend that you switch to the Fortran 95 formats for data files whenever possible.

By default, the **xlf90**, **xlf90_r**, and **xlf90_r7** commands do not conform completely to the Fortran 90 standard. Also, by default, the **xlf95**, **xlf95_r**, and **xlf95_r7** commands do not conform completely to the Fortran 95 standard. If you need full compliance, compile with any of the following additional compiler options (and suboptions):

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

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

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

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

Compiling XL Fortran SMP Programs

You can use the **xlf_r**, **xlf_r7**, **xlf90_r**, **xlf90_r7**, **xlf95_r**, or **xlf95_r7** command to compile XL Fortran SMP programs. The **xlf_r** and **xlf_r7** commands are similar to the **xlf** command, the **xlf90_r** and **xlf90_r7** commands are similar to the **xlf90** command, and the **xlf95_r** and **xlf95_r7** commands are similar to the **xlf95** command. The main difference is that the thread-safe components (libraries, **crt0_r.o**, and so on) are used to link and bind the object files if you specify the **xlf_r**, **xlf_r7**, **xlf90_r**, **xlf90_r7**, **xlf95_r**, or **xlf95_r7** command.

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

Levels of POSIX pthreads API Support

On AIX Version 5.1 and higher, XL Fortran supports 64-bit thread programming with the 1003.1-1996 (POSIX) standard pthreads API. It also supports 32-bit programming with both the Draft 7 and the 1003.1-1996 standard APIs.

You can use invocation commands (which use corresponding stanzas in the **xlf.cfg** configuration file) to compile and then link your programs with either the 1003.1-1996 standard or the Draft 7 interface libraries.

- To compile and then link your program with the 1003.1-1996 standard interface libraries, use the **xlf_r**, **xlf90_r**, or **xlf95_r** command. For example, you could specify:

  ```
  xlf95_r test.f
  ```

- To compile and then link your program with the Draft 7 interface libraries, use the **xlf_r7**, **xlf90_r7**, or **xlf95_r7** command. For example, you could specify:

  ```
  xlf95_r7 test.f
  ```

Apart from the level of thread support, the **xlf_r7**, **xlf90_r7**, and **xlf95_r7** commands and the corresponding stanzas in the **xlf.cfg** configuration file provide the same support as the **xlf_r**, **xlf90_r**, and **xlf95_r** commands and the corresponding stanzas.

Compilation Order for Fortran Programs

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

Canceling a Compilation

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

XL Fortran Input Files

The input files to the compiler are:

Source Files (.f or .F suffix)

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

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

Related Information: See "Passing Fortran Files through the C Preprocessor" on page 40.

The **fsuffix** and **cppsuffix** attributes in "Customizing the Configuration File" on page 15 and the "-qsuffix Option" on page 244 let you select a different suffix.

Object Files (.o suffix)

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

Related Information: See "Options That Control Linking" on page 86 and "Linking XL Fortran Programs" on page 42.

The **osuffix** attribute, which is described in "Customizing the Configuration File" on page 15 and the "-qsuffix Option" on page 244, lets you select a different suffix.

Assembler Source Files (.s suffix)

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

Related Information: The **ssuffix** attribute, which is described in "Customizing the Configuration File" on page 15 and the "-qsuffix Option" on page 244, lets you select a different suffix.

Archive or Library Files (.a suffix)
The compiler sends any specified library files (**.a** files) to the linker at link time. There are also AIX and XL Fortran library files in the **/usr/lib** directory that are linked in automatically.

Related Information: See "-l Option" on page 112, "-L Option" on page 111, and "LIBPATH:Setting Library Search Paths" on page 14.

Shared Object 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.

Related Information: See "-brtl Option" on page 100 and "-bdynamic, -bshared, and -bstatic Options" on page 95.

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

The default configuration file is **/etc/xlf.cfg**.

Related Information: See "Customizing the Configuration File" on page 15 and "-F Option" on page 107 for information about selecting the configuration file.

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

Related Information: See "-I Option" on page 109, "-qmoddir Option" on page 203, and "Displaying Information inside Binary Files (what)" on page 396.

Profile Data Files

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

Related Information: See "-qpdf Option" on page 210.

XL Fortran Output Files

The output files that XL Fortran produces are:

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

Related Information: See "-o Option" on page 116 for information on selecting a different name and "-c Option" on page 104 for information on generating only an object file.

Object Files: *filename***.o**

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

Related Information: See "-c Option" on page 104 and "Linking XL Fortran Programs" on page 42. For information on renaming the object file, see "-o Option" on page 116.

Assembler Source Files: *filename***.s**

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

Related Information: See "-S Option" on page 269 and "Linking XL Fortran Programs" on page 42. For information on renaming the assembler source file, see "-o Option" on page 116.

Compiler Listing Files: *filename***.lst**

By default, no listing is produced unless you specify one or more listing-related compiler options. The listing file is placed in the current directory, with the same file name prefix as the source file and an extension of **.lst**.

Related Information: See "Options That Control Listings and Messages" on page 77.

Module Symbol Files: *modulename***.mod**

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

Related Information: For information on putting **.mod** files in a different directory, see "-qmoddir Option" on page 203.

cpp-Preprocessed Source Files: F*filename***.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 40 and "-d Option" on page 106.

Profile Data Files (.*pdf*)

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

Related Information: See "-qpdf Option" on page 210.

Scope and Precedence of Option Settings

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

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

If you specify an option more than once with different settings, the last setting generally takes effect. Any exceptions are noted in the individual descriptions in the "Detailed Descriptions of the XL Fortran Compiler Options" on page 90 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
```

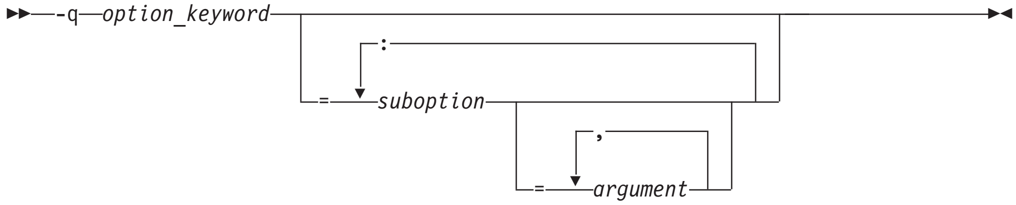
(There are some exceptions, such as **-pg**, which is a single option and not the same as **-p -g**.)

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″ Commands

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

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

  ```
  xlf95 -berok file.f # -berok is passed to ld
  ```
- Use the **-W** compiler option to construct an argument list for the command:

  ```
  xlf95 -Wl,-berok file.f # -berok is passed to ld
  ```

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

  This form is more general than the previous one because it works for the **as** command and any other commands called during compilation, by using different letters after the **-W** option.
- Edit the configuration file **/etc/xlf.cfg**, or construct your own configuration file. You can customize particular stanzas to allow specific command-line options to be passed through to the assembler or linker.

  For example, if you include these lines in the **xlf95** stanza of **/etc/xlf.cfg**:

  ```
  asopt = "w"
  ldopt = "m"
  ```

  and issue this command:

  ```
  xlf95 -wm -Wa,-x -Wl,-s produces_warnings.s uses_many_symbols.f
  ```

  the file **produces_warnings.s** is assembled with the options **-w** and **-x** (issue warnings and produce cross-reference), and the object files are linked with the options **-m** and **-s** (write list of object files and strip final executable file). .

Related Information: See “-W Option” on page 275 and “Customizing the Configuration File” on page 15.

Tracking Use of the Compiler

For customers who need to audit the use of the compiler, the XL Fortran compiler can be license management (LM) controlled using LUM (License Use Management). This was previously known as the NetLS** / iFOR/LS** product.

The system administrator can track the number of concurrent users who are logged onto a set of client machines. The compiler has a default of LM enabled, and all features of LUM will be available.

LUM can be disabled using the **-qnolm** compiler option. Use this option on the command line to disable LUM during a specific compile, or place the option in your config file (**xlf.cfg**) if you want LUM disabled by default.

The XLF software license allows a specific number of users to operate the compiler. LM, which is on by default, tracks the number of users if the enabling password has been installed as described in *Using LUM Runtime* and the *XL Fortran Installation Guide* accompanying XL Fortran Version 9.1.

Depending on the way XL Fortran users are distributed across a network, you may want to use *concurrent network licenses*, *concurrent nodelock licenses*, or a combination of both:

Concurrent network licenses
: Available to any authorized user on any machine in an LUM "cell". Depending on your configuration, they may require that the LUM client software be running on the same machine as the compiler. They can result in performance overhead during compilation.

  Users can be denied access to the compiler depending upon the authority provided by their user ID.

Concurrent nodelock licenses
: Restricted to a single machine, but they do not require the LUM client software or impose as much compilation performance overhead as concurrent network licenses.

  Users can be denied access to the compiler depending upon the authority provided by their user ID.

Related Information: See "-qlm Option" on page 197, the *Using LUM User's Guide*, and the *Using LUM Runtime*.

Compiling for Specific Architectures

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

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

If the compiling machine is also the target architecture, **-qarch=auto** will automatically detect the setting for the compiling machine. For more information on this compiler option setting, see "-qarch Option" on page 127 and also **-O4** and **-O5** under the "-O Option" on page 114.

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

Passing Fortran Files through the C Preprocessor

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

XL Fortran can use **cpp** to preprocess a file before compiling it. If you are also using one of the optimizing preprocessors, **cpp** is called before any of the other preprocessors.

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

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

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

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

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

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

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

```
!$omp end do
!$omp end parallel

     end
```

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

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

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

Related Information: See "-d Option" on page 106, "-t Option" on page 270, "-W Option" on page 275, and "Customizing the Configuration File" on page 15.

cpp Directives for XL Fortran Programs

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

Passing Options to the C Preprocessor

Because the compiler does not recognize **cpp** options other than **-I** directly on the command line, you must pass them through the **-W** option. For example, if a program contains **#ifdef** directives that test the existence of a symbol named **AIXV4**, you can define that symbol to **cpp** by compiling with a command like:

```
xlf95 conditional.F -WF,-DAIXV4
```

Avoiding Preprocessing Problems

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

Consider the following example:

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

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

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

On the AIX operating system, **cpp** has the option **-qlanglvl=classic**. Therefore, compile the trigraph example by using the following command:

```
xlf95 tst.F -d -v -WF,-qlanglvl=classic
```

This invokes **cpp tst.F -qlanglvl=classic**.

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 48 to execute the program.

Compiling and Linking in Separate Steps

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

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

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

Linking 32–Bit SMP Object Files Using the ld Command

To use the **ld** command to link an SMP program, follow these guidelines:

- Do not specify the **-e** flag.
- Do not change the default starting point of the executable output file (`__start`). If you use other starting points, your results will be unpredictable.
- Specify the following options and files with the **ld** command:
  - **-bh:4**, **-bpT:0x10000000**, **-bpD:0x20000000**.
  - **-lxlf** before any other libraries or files on the command line if you are linking any object files compiled by XL Fortran Version 2.
  - The object file that contains the system startup routine:
    - **crt0_r.o** for a program that was not profiled.
    - **mcrt0_r.o** for a program that was profiled with the **-p** option.
    - **gcrt0_r.o** for a program that was profiled with the **-pg** option.
    - Link with the startup files in **/usr/lib**.
  - Compiler and system libraries, in the following order:
    - **-lxlfpthrds_compat** (for POSIX pthreads Draft 7 support), **-lxlf90_r**, **-lxlf**, **-lxlsmp**, **-lm_r**, **-lc_r**, **-lc**, and either:
      - **-lpthreads** (for POSIX pthreads 1003.1-1996 standard support).
      - **-lpthreads_compat** followed by **-lpthreads** (for POSIX pthreads Draft 7 support).
    - You only need to specify **-lxlsmp** if you are compiling with **-qsmp**.

- If you use the **-qautodbl** option, specify some extra libraries that are listed in "-qautodbl Option" on page 134.
- If you use the **-qpdf1** compiler option, specify **-lxlopt**.
- If you use the **-qhot=vector** suboption, specify **-lxlopt**.

On AIX Version 5.1 and higher, the default POSIX pthreads API is the 1003.1-1996 standard. If you had a program called **mytest** and you wanted to obtain access to the functions in the 1003.1-1996 standard POSIX pthreads API on AIX Version 5.1, you would link with the **libpthreads.a** library, using something similar to the following command:

```
ld -bh:4 -bpT:0x10000000 -bpD:0x20000000 /lib/crt0_r.o mytest.o -lxlf90_r
   -lxlf -lxlsmp -lm_r -lm -lc_r -lc -lpthreads -o mytest
```

The 1003.1-1996 standard is not fully compatible with Draft 7. If you have programs that require the Draft 7 interface, link your programs with the **libpthreads_compat.a** and **libxlfpthrds_compat.a** libraries (which provide compatibility support) followed by the **libpthreads.a** library. For example, if you have a program called **mytest** that was written to use the Draft 7 interface, on AIX Version 5.1, you would use something similar to the following command:

```
ld -bh:4 -bpT:0x10000000 -bpD:0x20000000 /lib/crt0_r.o mytest.o
   -lxlfpthrds_compat -lxlf90_r -lxlf -lxlsmp -lm_r -lm -lc_r -lc
   -lpthreads_compat -lpthreads -o mytest
```

The configuration file **/etc/xlf.cfg** lists these default libraries and linker options. By doing a sample compilation with the **-#** option, you can see exactly how the compiler would run the linker.

See the *AIX Commands Reference* for a description of the linker options.

Linking 64–Bit SMP Object Files Using the ld Command

To use the **ld** command to link a 64-bit SMP program, follow these guidelines:

- Do not specify the **-e** flag.
- Do not change the default starting point of the executable output file (`__start`). If you use other starting points, your results will be unpredictable.
- Specify the following options and files with the **ld** command:
  – On AIX 5.1, **-bh:4**, **-bpT:0x10000000**, **-bpD:0x20000000**, **-b64**.
  – The object file that contains the system startup routine:
    - **crt0_64.o** for a program that was not profiled.
    - **mcrt0_64.o** for a program that was profiled with the **-p** option.
    - **gcrt0_64.o** for a program that was profiled with the **-pg** option.
  – Link with the startup files in **/usr/lib**.
  – Compiler and system libraries:
    - **-lxlf90**, **-lxlsmp**, **-lm**, **-lc**, and **-lpthreads**, in that order (you only need **-lxlsmp** if you compile with the **-qsmp** option).
    - If you use the **-qautodbl** option, specify some extra libraries that are listed in the "-qautodbl Option" on page 134.
    - If you use the **-qpdf1** compiler option, specify **-lxlopt**.
    - If you use the **-qhot=vector** suboption, specify **-lxlopt**.

For example, to link the object files `smpfile1.o` and `smpfile2.o` on AIX 5.1, you could specify the following:

```
ld -bh:4 -bpT:0x10000000 -bpD:0x20000000 -b64 /lib/crt0_64.o -lxlf90
   -lxlsmp -lm -lc smpfile1.o smpfile2.o
```

The configuration file **/etc/xlf.cfg** lists these default libraries and linker options. By doing a sample compilation with the **-#** option, you can see exactly how the compiler would run the linker.

See the *AIX Commands Reference* for a description of the linker options.

Linking 32–Bit Non-SMP Object Files Using the ld Command

To use the **ld** command to link non-SMP object files in a 32-bit environment, follow these guidelines:

- Do not specify the **-e** flag.
- Do not change the default starting point of the executable output file (`__start`). If you use other starting points, your results will be unpredictable.
- Specify the following options and files with the **ld** command:
  – **-bh:4**, **-bpT:0x10000000**, **-bpD:02x0000000**.
  – **-lxlf** before any other libraries or files on the command line if any object files compiled by XL Fortran Version 2 are being linked.
  – The object file that contains the system startup routine:
    - **crt0.o** for a program that was not profiled.
    - **mcrt0.o** for a program that was profiled with the **-p** option.
    - **gcrt0.o** for a program that was profiled with the **-pg** option.
    - Link with the startup files in **/usr/lib**.
  – Compiler and system libraries:
    - **-lxlf90**, **-lm**, and **-lc** in that order.
    - If you use the **-qautodbl** option, specify some extra libraries that are listed in “-qautodbl Option” on page 134.
    - If you use the **-qpdf1** compiler option, specify **-lxlopt**.
    - If you use the **-qhot=vector** suboption, specify **-lxlopt**.

For example, to link the object files `file1.o` and `file2.o`, you could specify the following:

```
ld -bh:4 -bpT:0x10000000 -bpD:0x20000000 /lib/crt0.o -lxlf90 -lm -lc
   file1.o file2.o
```

The configuration file **/etc/xlf.cfg** lists these default libraries and linker options. By doing a sample compilation with the **-#** option, you can see exactly how the compiler would run the linker.

See the *AIX Commands Reference* for a description of the linker options.

Linking 64-Bit Non-SMP Object Files Using the ld Command

To use the **ld** command to link non-SMP object files in a 64-bit environment, follow these guidelines:

- Do not specify the **-e** flag.
- Do not change the default starting point of the executable output file (`__start`). If you use other starting points, your results will be unpredictable.
- Specify the following options and files with the **ld** command:
  – On AIX 5.1, **-bh:4**, **-bpT:0x10000000**, **-bpD:0x20000000**, **-b64**.
  – The object file that contains the system startup routine:
    - **crt0_64.o** for a program that was not profiled.
    - **mcrt0_64.o** for a program that was profiled with the **-p** option.
    - **gcrt0_64.o** for a program that was profiled with the **-pg** option.
    - Link with the startup files in **/usr/lib**.
  – Compiler and system libraries:
    - **-lxlf90**, **-lm**, and **-lc** in that order.

- If you use the **-qautodbl** option, specify some extra libraries that are listed in "-qautodbl Option" on page 134.
- If you use the **-qpdf1** compiler option, specify **-lxlopt**.
- If you use the **-qhot=vector** suboption, specify **-lxlopt**.

For example, to link the object files `file1.o` and `file2.o` on AIX 5.1, you could specify the following:

```
ld -bh:4 -bpT:0x10000000 -bpD:0x20000000 -b64 /lib/crt0_64.o -lxlf90 -lm
   -lc file1.o file2.o
```

The configuration file **/etc/xlf.cfg** lists these default libraries and linker options. By doing a sample compilation with the **-#** option, you can see exactly how the compiler would run the linker.

See the *AIX Commands Reference* for a description of the linker options.

Passing Options to the ld Command

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

```
xlf95 -bhalt:2 -K -r file.f # xlf95 passes all these options to ld
```

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

Checking for Interface Errors at Link Time

If you specify the **-qextchk** compiler option, the linker may refuse to link object files containing mismatching procedure interfaces or common block definitions, allowing you to find these errors at link time, instead of trying to debug incorrect results.

If the linking problem can be isolated to a few names that do not resolve, perhaps because of uppercase letters in C names or trailing underscores added by the **-qextname** option, you can use the **-brename** linker option to change just those names:

```
xlf95 -brename:Old_Link_Name,new_link_name fort_prog.o c_prog.o
```

Related Information: See "-qextchk Option" on page 156, "-U Option" on page 271, and "-qextname Option" on page 158.

Linking New Objects with Existing Ones

If you have **.o** or other object files that you compiled with an earlier version of XL Fortran, you can link them with object files that you compile with XL Fortran Version 9, subject to the following notes. The main XL Fortran libraries are **libxlf90.a** and **libxlf90_r.a**, but calls to older entry points in **libxlf.a** are still possible; the calls are passed to the new entry points in the main libraries, which makes the resulting programs slower than if everything is recompiled.

Notes:

1. You must explicitly specify the XL Fortran **libxlf.a** library as part of the link step, preferably by including the option **-lxlf**.
2. For safety, always put **-lxlf** as the first option after the compiler command so that the library is linked before any user object files. Doing so ensures that the new I/O routines override any existing ones in statically linked object files.
3. When you relink old object files, the I/O routines in the resulting program differ in some ways from the behavior of XL Fortran Version 2. To make the

resulting program work as you expect, you may need to change some of the run-time settings in "Setting Run-Time Options" on page 51 (particularly the **namelist** setting) or to recompile the source files with the "-qxlf77 Option" on page 261. Some changed I/O details cannot be switched to the old behavior at all.

4. You cannot link files that you compiled with the XL Fortran Version 4 level of IPA with files that you compiled with the XL Fortran Version 6 level or later of IPA.
5. You cannot link 64-bit objects compiled with XL Fortran version 7.1.0.1, or lower. The object format has changed on AIX Version 5.1.
6. You cannot link **pdf** files that you created with **-qpdf1** and Version 5.1.0 or earlier levels of XL Fortran with **pdf** files that you created with **-qpdf1** and XL Fortran Version 7.1 or higher. However, you can link object files that you created with **-qpdf2** and XL Fortran Version 7.1 or higher with object files that you created with **-qpdf2** and earlier levels of XL Fortran.

Relinking an Existing Executable File

Because the linker accepts executable files as input, you can link an existing executable file with updated object files. You cannot, however, relink executable files that were previously linked using the **-qipa** option.

If you have a program consisting of several source files and only make localized changes to some of the source files, you do not necessarily have to compile each file again. Instead, you can include the executable file as the last input file when compiling the changed files:

```
xlf95 -omansion front_door.f entry_hall.f parlor.f sitting_room.f \
  master_bath.f kitchen.f dining_room.f pantry.f utility_room.f

vi kitchen.f # Fix problem in OVEN subroutine

xlf95 -o newmansion kitchen.f mansion
```

Limiting the number of files to compile and link the second time reduces the compile time, disk activity, and memory use.

Note: If this type of linking is done incorrectly, it can result in interface errors and other problems. Therefore, you should not try it unless you are experienced with linking.

Dynamic and Static Linking

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

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

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

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

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

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

  You can use **-b** linker options on the compiler command line to create statically linked object files:

  ```
  xlf95 -bnso -bI:/usr/lib/syscalls.exp file1.f file2.f
  ```

  You must also specify **-bI:/usr/lib/threads.exp** when you are statically linking with the **xlf_r**, **xlf_r7**, **xlf90_r**, **xlf90_r7**, **xlf95_r**, or **xlf95_r7** command.

  If you are using Asynchronous I/O, you must also specify **-bI:/usr/lib/aio.exp**.

  The **-bnso** option places the library procedures that your program references into the program's object file. Files with a suffix of **.exp** specify the names of system routines that must be imported to your program from the system.

  An alternative that requires less disk space is to link any XL Fortran libraries statically but to leave references to other system libraries dynamic. This example statically links just the XL Fortran libraries:

  ```
  # Build a temporary object from the Fortran library:
     ld -r -o libtmp.o -bnso -lxlf90
  # Build the application with this object on the command line:
     xlf95 -o appl appl1.o appl2.o libtmp.o
  ```

Avoiding Naming Conflicts during Linking

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

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

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

  Note: When you use this option, you do not need to use the final underscore in the names of Service and Utility Subprograms, such as **dtime_** and **flush_**.
- Link your programs dynamically, which is the default. Many naming conflicts only apply to statically linked programs.
- Order the names of libraries and object files on the command line so that the one that should take precedence comes first. For example, to make names in **libxlf90.a** take precedence over duplicate names in an object file, specify **-lxlf90** first on the command line.

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

- Do not name a subroutine or function **main**, because XL Fortran defines an entry point **main** to start your program.
- Do not use *any* global names that begin with an underscore. In particular, the XL Fortran libraries reserve all names that begin with **_xl**.

- Do not use names that are the same as names in the XL Fortran library or one of the system libraries. To determine which names are not safe to use in your program, run the **nm** command on any libraries that are linked into the program and search the output for names you suspect might also be in your program.
- If your program calls certain XLF-provided routines, some restrictions apply to the common block and subprogram names that you can use:

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

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

If different versions of a routine occur in more than one library or object file, be careful to use the specific version that you want. Specify the file with the correct version as the first file on the command line or in the configuration file. If the file is a library, specify the appropriate **-l** option first on the command line. This technique does not apply to references between routines that are in the same shared library or to routines that are explicitly imported from one shared library to another.

Running XL Fortran Programs

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

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

Canceling Execution

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

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

Running Previously Compiled Programs

Statically linked programs that you compiled with levels of XL Fortran prior to Version 9.1 should continue to run with no change in performance or behavior. They may not run on a system with a level of the operating system different from the system on which they were compiled.

If you have dynamically linked programs compiled by XL Fortran Versions 2 through 8, you can run them on systems with the XL Fortran Version 9 libraries. The programs will use the current compiler data formats and I/O behavior, which are somewhat different from those of XL Fortran Version 2.

Compiling and Executing on Different Systems

If you want to move an XL Fortran executable file to a different system for execution, you can link statically and copy the program, and optionally the run-time message catalogs. Alternatively, you can link dynamically and copy the program as well as the XL Fortran libraries if needed and optionally the run-time message catalogs. For non-SMP programs, **libxlf90.a** is usually the only XL Fortran library needed. For SMP programs, you will usually need at least the **libxlf90_r.a** and **libxlsmp.a** libraries. **libxlf.a** is only needed if the program has any XL Fortran Version 1 or 2 object files linked in. **libxlfpmt*.a** and **libxlfpad.a** are only needed if the program is compiled with the **-qautodbl** option. If your application has dependencies on **libhmd.a**, refer to "Using Debug Memory Routines for XL Fortran" on page 381 for more details on library dependencies.

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 46.

POSIX Pthreads Binary Compatibility

The XL Fortran compiler and run-time library provide binary compatibility in the following areas:

- Executable file binary compatibility. If you created an executable file that had dependencies on the pthreads Draft 7 API (for example, you used XL Fortran Version 5.1.0 or AIX Version 4.2.1), you can upgrade your system to use XL Fortran Version 9.1.0 or AIX Version 5.1 and run your executable file without first recompiling and relinking your program.
- Object file or archive library binary compatibility. If you created an object file or archive library that had dependencies on the Draft 7 pthreads API, you can continue to use that object file or archive library with the Draft 7 interface if you move from AIX Version 4.2.1 to AIX Version 5.1. For example, if you have a source file called **test.f** that uses a shared or static archive library called **libmy_utility.a** (which was created with the Draft 7 interface), you would enter something similar to the following command on AIX Version 5.1:

  ```
  xlf95_r7 test.f -lmy_utility -o a.out
  ```

  You do not need to regenerate **libmy_utility.a** before using it on AIX Version 5.1.

There are, however, restrictions on binary compatibility. XL Fortran supports combinations of Draft 7 and 1003.1-1996 standard object files in some instances. For example, if you used XL Fortran Version 5.1.0 to create a library, that library uses the Draft 7 pthreads API. An application that you build with that library can use either the Draft 7 pthreads API or the 1003.1-1996 standard pthreads API as long as the portions of the complete application built with the Draft 7 pthreads API do not share any pthreads data objects (such as mutexes or condition variables) with the portions built with the 1003.1-1996 standard pthreads API. If any such objects need to be used across portions of an application that are compiled with different levels of the pthreads API, the final application needs to use either the Draft 7 pthreads API or the 1003.1-1996 standard pthreads API across the entire application. You can do this in one of two ways:

- Build the application by using the **xlf_r7**, **xlf90_r7**, or **xlf95_r7** command, so that it uses the Draft 7 pthreads API.
- Build both the library and the rest of the application by using the **xlf_r**, **xlf90_r**, or **xlf95_r** command.

Run-Time Libraries for POSIX Pthreads Support

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

The following libraries are used:

/lib/libxlf90.a
: Provides 1003.1-1996 standard 32-bit and 64-bit support. This library is linked to **libxlf90_r.a**.

/lib/libxlsmp.a
: Provides 1003.1-1996 standard 32-bit and 64-bit support.

/lib/libxlfpthrds_compat.a
: Provides Draft 7 32-bit support.

XL Fortran supplies the following directories for **.mod** files:

/usr/lpp/xlf/include_d7
: Provides Draft 7 32-bit support.

/usr/lpp/xlf/include
: Provides 1003.1-1996 standard 32-bit and 64–bit support.

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

Cmd.	Libraries Used	Include Files Used	POSIX Pthreads API Level Supported
xlf90_r **xlf95_r** **xlf_r**	**/lib/libxlf90.a** **/lib/libxlsmp.a** **/lib/libpthreads.a**	**/usr/lpp/xlf/include**	1003.1-1996 standard
xlf90_r7 **xlf95_r7** **xlf_r7**	**/lib/libxlf90.a** **/lib/libxlsmp.a** **/lib/libxlfpthrds_compat.a** **/lib/libpthreads.a**	**/usr/lpp/xlf/include_d7**	Draft 7

Selecting the Language for Run-Time Messages

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

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

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

Related Information: See "Environment Variables for National Language Support" on page 13.

The C library routine **setlocale** is defined in the *AIX Technical Reference: Base Operating System and Extensions Volume 1*.

Setting Run-Time Options

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

The XLFRTEOPTS Environment Variable

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

```
                         ┌─:────────────────────────────────────────────┐
                         ↓                                              │
►►──XLFRTEOPTS=──┬─────┬──runtime_option_name──=────option_setting──┴──┬─────┬──────►◄
                 └─"───┘                                                └─"───┘
```

You can specify option names and settings in uppercase or lowercase. You can add blanks before and after the colons and equal signs to improve readability. However, if the **XLFRTEOPTS** option string contains imbedded blanks, you must enclose the entire option string in double quotation marks (**"**).

The environment variable is checked when the program first encounters one of the following conditions:
- An I/O statement is executed.
- The **RANDOM_SEED** procedure is executed.
- An **ALLOCATE** statement needs to issue a run-time error message.
- A **DEALLOCATE** statement needs to issue a run-time error message.
- The multi-threaded implementation of the **MATMUL** procedure is executed.

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

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

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

buffering={enable | disable_preconn | disable_all}
Determines whether the XL Fortran run-time library performs buffering for I/O operations.

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

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

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

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

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

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

disable_preconn
The Fortran run-time library does not maintain an I/O buffer for each preconnected logical unit (0, 5, and 6). However, it does maintain I/O buffers for all other connected logical units. The current read-write file pointers that the run-time library maintains for the preconnected units are the same as the read-write pointers of the corresponding files in the file system.

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

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

Fortran main program:

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

C subroutine (c_func.c):

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

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

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.

cnverr={yes | no}

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

Related Information: For more information about conversion errors, see *Data Transfer Statements* in the *XL Fortran Enterprise Edition for AIX Language Reference*. For more information about **IOSTAT** values, see *Conditions and IOSTAT Values* in the *XL Fortran Enterprise Edition for AIX 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. (For a definition of user time, see the *AIX Performance Management Guide*).

systime

Returns the system time of a process. (For a definition of system time, see the *AIX Performance Management Guide*).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

langlvl={extended | 90ext| 90std | 95std | 2003std}
Determines the level of support for Fortran standards and extensions to the standards. The values of the suboptions are as follows:

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

90ext
Currently, provides the same level of support as the **extended** suboption. **90ext** was the default suboption prior to XL Fortran Version 7.1. However, this suboption is now obsolete, and to avoid problems in the future, you should start using the **extended** suboption as soon as possible.

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

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

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

```
integer(4) aa(100)
call setrteopts("langlvl=2003std")
    ...          ! Write to a unit without explicitly
    ...          ! connecting the unit to a file.
write(10, *) aa  ! The implicit connection to a file does not
    ...          ! comform 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 as of Version 9.1 (such as namelist comments), you must specify one of the following suboptions:

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

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

```
program test1

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

10 format(I3)

write(1,fmt=10) 123

end
```

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

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

```
program test2

INTEGER I
LOGICAL G
NAMELIST /TODAY/G, I

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

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

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

end

today.new:

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

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

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

multconn={yes | <u>no</u>}

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, and tape drives)

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 (Version 1) **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-compilant new format in writing any new data files.

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

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

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

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

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

scratch_vars={yes | no}
To give a specific name to a scratch file, set the **scratch_vars** run-time option to **yes**, and set the environment variable **XLFSCRATCH_***unit* to the name of the file you want to be associated with the specified unit number. See "Naming Scratch Files" on page 332 for examples.

unit_vars={yes | no}
To give a specific name to an implicitly connected file or to a file opened with no **FILE=** specifier, you can set the run-time option **unit_vars=yes** and set one or more environment variables with names of the form **XLFUNIT_***unit* to file names. See "Naming Files That Are Connected with No Explicit Name" on page 331 for examples.

uwidth={32 | 64}
To specify the width of record length fields in unformatted sequential files, specify the value in bits. When the record length of an unformatted sequential file is greater than (2**31 - 1) bytes minus 8 bytes (for the record terminators surrounding the data), you need to set the run-time option **uwidth=64** to extend the record length fields to 64 bits. This allows the record length to be up to (2**63 - 1) minus 16 bytes (for the record terminators surrounding the data). The run-time option **uwidth** is only valid for 64-bit mode applications.

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

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

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

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

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

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

The XLSMPOPTS Environment Variable

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

```
                         ┌─:──────────────────────────────────────────┐
                         ▼                                            │
►►──XLSMPOPTS=──┬─────┬──runtime_option_name──=──option_setting──┴──┬─────┬──────►◄
                └─"───┘                                              └─"───┘
```

You can specify option names and settings in uppercase or lowercase. You can add blanks before and after the colons and equal signs to improve readability. However, if the **XLSMPOPTS** option string contains imbedded blanks, you must enclose the entire option string in double quotation marks (**"**).

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

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

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

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

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

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

```
CEILING(number_of_iterations / number_of_threads)
```

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

Choosing chunking granularity is a tradeoff between overhead and load balancing. The syntax for this option is **schedule**=*suboption*, where the suboptions are defined as follows:

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

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

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

guided[=*n***]**
: The iterations of a loop are divided into progressively smaller chunks until a minimum chunk size of *n* loop iterations is reached. If you have not specified *n*, the default value for *n* is 1 iteration.

  The first chunk contains CEILING(number_of_iterations / number_of_threads) iterations. Subsequent chunks consist of CEILING(number_of_iterations_left / number_of_threads) iterations.

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

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

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

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

Parallel execution options

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

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

	used for parallel execution of code that you compiled with the **-qsmp** option. By default, this is equal to the number of online processors. There are some applications that cannot use more than some maximum number of processors. There are also some applications that can achieve performance gains if they use more threads than there are processors. This option allows you full control over the number of execution threads. The default value for *num* is 1 if you did not specify **-qsmp**. Otherwise, it is the number of online processors on the machine. For more information, see the **NUM_PARTHDS** intrinsic function in the *XL Fortran Enterprise Edition for AIX Language Reference*.
usrthds=*num*	Specifies the maximum number of threads (*num*) that you expect your code will explicitly create if the code does explicit thread creation. The default value for *num* is 0. For more information, see the **NUM_PARTHDS** intrinsic function in the *XL Fortran Enterprise Edition for AIX Language Reference*.
stack=*num*	Specifies the largest amount of space in bytes (*num*) that a thread's stack will need. The default value for *num* is 4194304. Set **stack**=*num* so it is within the acceptable upper limit. *num* can be up to 256 MB for 32-bit mode, or up to the limit imposed by system resources for 64-bit mode. An application that exceeds the upper limit may cause a segmentation fault.
startproc=*cpu_id*	Specifies the CPU ID that the first thread should be bound to. If the value provided is less than 0 (zero), the SMP run time issues a warning message and no threads are bound.
stride=*num*	*num* specifies the number of processors to skip. This must be greater than or equal to 1. If the value provided is less than 1, a warning message is issued and no threads are bound.

Performance tuning options

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

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

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

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

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

The syntax for specifying these options is as follows:

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

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

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

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

For instance, on a dedicated 8-way SMP, setting these options to the following:

```
parthds=8 : schedule=dynamic=10 : spins=0 : yields=0
```

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

You can also use the environment variables **SPINLOOPTIME** and **YIELDLOOPTIME** to tune performance. Refer to the *AIX Performance Management Guide* for more information on these variables.

Options to enable and control dynamic profiling

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

parthreshold=*num*
: Specifies the time, in milliseconds, below which each loop must execute serially. If you set **parthreshold** to 0, every loop that has been parallelized by the compiler will execute in parallel. The default setting is 0.2 milliseconds, meaning that if a loop requires fewer than 0.2 milliseconds to execute in parallel, it should be serialized.

Typically, **parthreshold** is set to be equal to the parallelization overhead. If the computation in a parallelized loop is very small and the time taken to execute these loops is spent primarily in the setting up of parallelization, these loops should be executed sequentially for better performance.

seqthreshold=*num*
: Specifies the time, in milliseconds, beyond which a loop that was previously serialized by the dynamic profiler should revert to being a parallel loop. The default setting is 5 milliseconds, meaning that if a loop requires more than 5 milliseconds to execute serially, it should be parallelized.

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

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

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

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

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

OpenMP Environment Variables

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

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

OMP_DYNAMIC Environment Variable

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

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

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

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

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

OMP_NESTED Environment Variable

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

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

If you set this environment variable to **TRUE**, nested parallelism is enabled. This means that the run-time environment might deploy extra threads to form the team of threads for the nested parallel region. If you set this environment variable to **FALSE**, nested parallelism is disabled.

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

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

OMP_NUM_THREADS Environment Variable

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

```
►►──OMP_NUM_THREADS=──num──────────────────────────────────────────────►◄
```

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

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

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

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

```
export OMP_NUM_THREADS=16
```

OMP_SCHEDULE Environment Variable

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

```
>>--OMP_SCHEDULE=--sched_type--+----------------+------------------><
                               '-,--chunk_size--'
```

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

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

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

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

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

It is set to the following for the Nth thread, where `N` is the total number of threads and `Iters` is the total number of iterations in the **DO** loop:

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

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

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

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

Other Environment Variables That Affect Run-Time Behavior

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

XL Fortran Run-Time Exceptions

The following operations cause run-time exceptions in the form of **SIGTRAP** signals, which typically result in a “Trace/BPT trap” message:
- Fixed-point division by zero.
- 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 exceptions occur after you specify the appropriate **-qflttrap** suboptions at compile time.
- Floating-point operations that generate NaN values and loads of the NaN values after you specify the **-qfloat=nanq** option 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. **dbx** provides a specific error message that describes the cause of the exception.

Related Information: See “-C Option” on page 103, “-qflttrap Option” on page 165, and “-qsigtrap Option” on page 232.

See “Detecting and Trapping Floating-Point Exceptions” on page 296 for more details about these exceptions and “Controlling the Floating-Point Status and Control Register” on page 299 for a list of exception handlers.

XL Fortran Compiler-Option Reference

This section contains the following:

- Tables of compiler options. These tables are organized according to area of use and contain high-level information about the syntax and purpose of each option.
- Detailed information about each compiler option in "Detailed Descriptions of the XL Fortran Compiler Options" on page 90.

Summary of the XL Fortran Compiler Options

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

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

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

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

Options That Control Input to the Compiler

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

Related Information: See "XL Fortran Input Files" on page 33 and "Options That Specify the Locations of Output Files" on page 70.

Many of the options in "Options for Compatibility" on page 79 change the permitted input format slightly.

Table 3. Options That Control Input to the Compiler

Command-Line Option	@PROCESS Directive	Description	See Page
`-I`*dir*		Adds a directory to the search path for include files and **.mod** files. If XL Fortran calls **cpp**, this option adds a directory to the search path for **#include** files. Before checking the default directories for include and **.mod** files, the compiler checks each directory in the search path. For include files, this path is only used if the file name in an **INCLUDE** line is not provided with an absolute path. For **#include** files, refer to the **cpp** documentation for the details of the `-I` option. **Default:** The following directories are searched, in the following order: 1. The current directory 2. The directory where the source file is 3. **/usr/include**.	109
-qci=*numbers*	CI(*numbers*)	Activates the specified **INCLUDE** lines. **Default:** No default value.	141
-qcr -qnocr		Allows you to control how the compiler interprets the CR (carriage return) character. This allows you to compile code written using a Mac OS or DOS/Windows editor. **Default:** **-qcr**	143

Table 3. Options That Control Input to the Compiler (continued)

Command-Line Option	@PROCESS Directive	Description	See Page
-qdirective [=*directive_list*] -qnodirective [=*directive_list*]	DIRECTIVE [(*directive_list*)] NODIRECTIVE [(*directive_list*)]	Specifies sequences of characters, known as trigger constants, that identify comment lines as compiler comment directives. **Default:** Comment lines beginning with **IBM*** are considered directives. If you specify **-qsmp=omp**, only **$OMP** is considered to be a directive trigger. All other directive triggers are turned off unless you explicitly turn them back on. If you specify **-qsmp=noomp** (noomp is the default for -qsmp), **IBMP**, **$OMP** and **SMP$** are considered directive triggers, along with any other directive triggers that are turned on (such as **IBM*** and **IBMT**). If you have also specified **-qthreaded**, comment lines beginning with **IBMT** are also considered directives.	148
-qfixed [=*right_margin*]	FIXED [(*right_margin*)]	Indicates that the input source program is in fixed source form and optionally specifies the maximum line length. **Default:** **-qfree=f90** for the **f90**, **f95**, **xlf90**, **xlf90_r**, **xlf90_r7**, **xlf95**, **xlf95_r**, and **xlf95_r7** commands and **-qfixed=72** for the **xlf**, **xlf_r**, **xlf_r7**, and **f77**/**fort77** commands.	161
-qfree[={f90 | ibm}] -k	FREE[({F90 | IBM})]	Indicates that the source code is in free form. The ibm and f90 suboptions specify compatibility with the free source form defined for VS FORTRAN and Fortran 90/Fortran 95, respectively. -k and -qfree are short forms for -qfree=f90. **Default:** **-qfree=f90** for the **f90**, **f95**, **xlf90**, **xlf90_r**, **xlf90_r7**, **xlf95**, **xlf95_r**, and **xlf95_r7** commands and **-qfixed=72** for the **xlf**, **xlf_r**, **xlf_r7**, and **f77**/**fort77** commands.	168
-qmbcs -qnombcs	MBCS NOMBCS	Indicates to the compiler whether character literal constants, Hollerith constants, H edit descriptors, and character string edit descriptors can contain Multibyte Character Set (MBCS) or Unicode characters. **Default:** **-qnombcs**	201
-U -qmixed -qnomixed	MIXED NOMIXED	Makes the compiler sensitive to the case of letters in names. **Default:** **-qnomixed**	271

Table 3. Options That Control Input to the Compiler (continued)

Command-Line Option	@PROCESS Directive	Description	See Page
-qsuffix={suboptions}		Specifies the source-file suffix on the command line.	244

Options That Specify the Locations of Output Files

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

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

Related Information: See "XL Fortran Output Files" on page 34 and "Options That Control Input to the Compiler" on page 68.

Table 4. Options That Specify the Locations of Output Files

Command-Line Option	@PROCESS Directive	Description	See Page
-d		Leaves preprocessed source files produced by **cpp**, instead of deleting them. **Default:** Temporary files produced by **cpp** are deleted.	106
-o *name**		Specifies a name for the output object, executable, or assembler source file. **Default:** -o a.out	116
-qmoddir=*directory*		Specifies the location for any module (**.mod**) files that the compiler writes. **Default:** **.mod** files are placed in the current directory.	203

Options for Performance Optimization

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

Related Information: See "Optimizing XL Fortran Programs" on page 305.

Some of the options in "Options for Floating-Point Processing" on page 86 can also improve performance, but you must use them with care to avoid error conditions and incorrect results.

Table 5. Options for Performance Optimization

Command-Line Option	@PROCESS Directive	Description	See Page
-O[*level*] -qoptimize[=*level*] -qnooptimize	OPTimize[(*level*)] NOOPTimize	Specifies whether code is optimized during compilation and, if so, at which level (0, 2, 3, 4, or 5). **Default:** **-qnooptimize**	114

Table 5. Options for Performance Optimization (continued)

Command-Line Option	@PROCESS Directive	Description	See Page
-P{v | k}[!]		Invokes the selected optimizing preprocessor. Adding **!** prevents the compilation step from following preprocessing. **Note:** The preprocessors are available as separate vendor-logo products. **Default:** No preprocessing.	117
-p -pg		Sets up the object file for profiling. **Default:** No profiling.	118
-Q -Q! -Q+*names* -Q-*names*		Specifies whether procedures are inlined and/or particular procedures that should or should not be inlined. *names* is a list of procedure names separated by colons. **Default:** No inlining.	119
-qalias= {[no]aryovrlp | [no]intptr | [no]pteovrlp | [no]std}...]	ALIAS({[NO]ARYOVRLP | [NO]INTPTR | [NO]PTEOVRLP | [NO]STD}...)	Indicates whether a program contains certain categories of aliasing. The compiler limits the scope of some optimizations when there is a possibility that different names are aliases for the same storage locations. **Default: -qalias=aryovrlp:nointptr:pteovrlp:std** for the **xlf90**, **xlf90_r7**, **xlf95**, **xlf95_r**, **xlf95_r7**, **f90**, and **f95** commands; **-qalias= aryovrlp:intptr:pteovrlp:std** for the **xlf**, **xlf_r**, **xlf_r7**, **f77**, and **fort77**, commands.	122
-qalign={[no]4k | struct {=subopt} | bindc {=subopt}}	ALIGN({[NO]4K | STRUCT{(subopt)} | BINDC{(subopt)}})	Specifies the alignment of data objects in storage, which avoids performance problems with misaligned data. The **[no]4k**, **bindc**, and **struct** options can be specified and are not mutually exclusive. The **[no]4k** option is useful primarily in combination with logical volume I/O and disk striping. **Default: -qalign=no4k:struct=natural:bindc=power**	125
-qarch=*architecture*		Controls which instructions the compiler can generate. Changing the default can improve performance but might produce code that can only be run on specific machines. The choices are auto, com, pwr, pwr2 (or pwrx), pwr2s, p2sc, 601, 603, 604, ppc, ppcgr, ppc64, ppc64gr, ppc64grsq, rs64a, rs64b, rs64c, pwr3, pwr4, pwr5, and ppc970. **Default: -qarch=com** if you specify **-q32**, which uses only instructions that are common to all platforms. If you specify **-q64**, the default is **ppc64**, which allows you to run the executable file on any 64-bit PowerPC hardware platform.	127
-qassert={ deps | nodeps | itercnt=*n*}		Provides information about the characteristics of the files that can help to fine-tune optimizations. **Default: -qassert=deps:itercnt=1024**	132

Table 5. Options for Performance Optimization (continued)

Command-Line Option	@PROCESS Directive	Description	See Page
-qcache={ auto | assoc=*number* | cost=*cycles* | level=*level* | line=*bytes* | size=*Kbytes* | type={C|c|D|d|I|i}}[:...]		Specifies the cache configuration for a specific execution machine. The compiler uses this information to tune program performance, especially for loop operations that can be structured (or *blocked*) to process only the amount of data that can fit into the data cache. **Default:** The compiler uses typical values based on the **-qtune** setting, the **-qarch** setting, or both settings.	137
-qcompact -qnocompact	COMPACT NOCOMPACT	Reduces optimizations that increase code size. **Default: -qnocompact**	142
-qdirectstorage -qnodirectstorage		Informs the compiler that a given compilation unit may reference write-through-enabled or cache-inhibited storage. Use this option with discretion. It is intended for programmers who know how the memory and cache blocks work, and how to tune their applications for optimal performance. **Default: -qnodirectstorage**	150
-qessl		Allows the use of ESSL routines in place of Fortran 90 Intrinsic Procedures. Use the ESSL Serial Library when linking with **-lessl**. Use the ESSL SMP Library when linking with **-lesslsmp**. **Default: -qnoessl**	155
-qfdpr -qnofdpr		Provides object files with information needed for theAIX fdpr (Feedback Directed Program Restructuring) performance-tuning utility to optimize the resulting executable file. **Default: -qnofdpr**	160
-qhot[=*suboptions*] -qnohot	HOT[=*suboptions*] NOHOT	The **-qhot** compiler option is a powerful alternative to hand tuning that provides opportunities to optimize loops and array language. The **-qhot** compiler option will always attempt to optimize loops, regardless of the suboptions you specify. **Default: -qnohot**	171
-qipa[=*suboptions*] | -qnoipa		Enhances **-O** optimization by doing detailed analysis across procedures (interprocedural analysis or IPA). **Default: -O** analyzes each subprogram separately, ruling out certain optimizations that apply across subprogram boundaries. Note that specifying **-O5** is equivalent to specifying **-O4** and **-qipa=level=2**.	182
-qkeepparm -qnokeepparm		Ensures that incoming procedure parameters are stored on the stack even when optimizing. **Default: -qnokeepparm**	188

Table 5. Options for Performance Optimization (continued)

Command-Line Option	@PROCESS Directive	Description	See Page
-qlargepage -qnolargepage		Indicates to the compiler that a program, designed to execute in a large page memory environment, can take advantage of large 16 MB pages provided on POWER4 and higher based systems. **Default: -qnolargepage**	191
-qmaxmem= *Kbytes*	MAXMEM (*Kbytes*)	Limits the amount of memory that the compiler allocates while performing specific, memory-intensive optimizations to the specified number of kilobytes. A value of -1 allows optimization to take as much memory as it needs without checking for limits. **Default: -qmaxmem=2048**; At **-O3**, **-O4**, and **-O5**, **-qmaxmem=-1**.	199
-qOBJect -qNOOBJect	OBJect NOOBJect	Specifies whether to produce an object file or to stop immediately after checking the syntax of the source files. **Default: -qobject**	207
-qpdf{1|2}		Tunes optimizations through *profile-directed feedback* (PDF), where results from sample program execution are used to improve optimization near conditional branches and in frequently executed code sections. **Default:** Optimizations use fixed assumptions about branch frequency and other statistics.	210
-qprefetch | -qnoprefetch		Indicates whether or not the prefetch instructions should be inserted automatically by the compiler. **Default: -qprefetch**	220
-qsaveopt -qnosaveopt		Saves the command-line options used for compiling a source file in the corresponding object file. **Default: -qnosaveopt**	229
-qshowpdf -qnoshowpdf		Adds additional call and block count profiling information to an executable. This option is used together with the **-qpdf1** option. **Default: -qnoshowpdf**	231
-qsmallstack -qnosmallstack		Specifies that the compiler will minimize stack usage where possible. **Default: -qnosmallstack**	233
-qsmp[=*suboptions*] -qnosmp		When used with **xlf_r**, **xlf_r7**, **xlf90_r**, **xlf90_r7**, **xlf95_r**, or **xlf95_r7**, controls automatic parallelization of loops, user-directed parallelization of loops and other items, and the choice of chunking algorithm. **Default: -qnosmp**	234

Table 5. Options for Performance Optimization (continued)

Command-Line Option	@PROCESS Directive	Description	See Page
-NS*bytes* -qSPILLsize= *bytes*	SPILLsize (*bytes*)	Specifies the size of internal program storage areas. **Default:** -NS512	113
-qstrict -qnostrict	STRICT NOSTRICT	Ensures that optimizations done by the -O3, -O4, -O5, -qhot, and -qipa options do not alter the semantics of a program. **Default:** With **-O3** and higher levels of optimization in effect, code may be rearranged so that results or exceptions are different from those in unoptimized programs. For **-O2**, the default is -**qstrict**. This option is ignored for **-qnoopt**.	241
-qstrictieeemod -qnostrictieeemod	STRICTIEEE-MOD NOSTRICTIEEE-MOD	Specifies whether the compiler will adhere to the Fortran 2003 IEEE arithmetic rules for the **ieee_arithmetic** and **ieee_exceptions** intrinsic modules. **Default: -qstrictieeemod**	242
-qstrict_induction -qnostrict_induction		Prevents the compiler from performing induction (loop counter) variable optimizations. These optimizations may be *unsafe* (may alter the semantics of your program) when there are integer overflow operations involving the induction variables. **Default: -qnostrict_induction**	243
-qthreaded		Specifies that the compiler should generate thread-safe code. This is turned on by default for the **xlf_r**, **xlf_r7**, **xlf90_r**, **xlf90_r7**, **xlf95_r**, and **xlf95_r7** commands.	250
-qtune=*implementation*		Tunes instruction selection, scheduling, and other implementation-dependent performance enhancements for a specific implementation of a hardware architecture. The following settings are valid: auto, pwr, pwr2 (or pwrx), pwr2s, p2sc, 601, 603, 604, rs64a, rs64b, rs64c, pwr3, pwr4, pwr5, or ppc970. **Default: -qtune=pwr2**, if you specify **-q32** and enable the **-qarch=com** option. If you specify **-q64** and enable the **-qarch=ppc** option, the default is **-qtune=pwr3**.	251
-qunroll [=auto | yes] -qnounroll		Specifies whether the compiler is allowed to automatically unroll **DO** loops. **Default: -qunroll=auto**	255
-qunwind -qnounwind	UNWIND NOUNWIND	Specifies default behavior for saving and restoring from non-volatile registers during a procedure call. **Default: -qunwind**	256
-qversion -qnoversion		Displays the version and release of the invoking compiler. **Default: -qnoversion**	257

Table 5. Options for Performance Optimization (continued)

Command-Line Option	@PROCESS Directive	Description	See Page
-qzerosize -qnozerosize	ZEROSIZE NOZEROSIZE	Improves performance of FORTRAN 77 and some Fortran 90 and Fortran 95 programs by preventing checking for zero-sized character strings and arrays. **Default: -qzerosize** for the **xlf90**, **xlf90_r**, **xlf90_r7**, **xlf95**, **xlf95_r**, **xlf95_r7**, **f90**, and **f95** commands and **-qnozerosize** for the **xlf**, **xlf_r**, **xlf_r7**, **f77**, and **fort77** commands (meaning these commands cannot be used for programs that contain zero-sized objects).	268

Options for Error Checking and Debugging

The following options help you avoid, detect, and correct problems in your XL Fortran programs and can save you having to refer as frequently to "Problem Determination and Debugging" on page 369.

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

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

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

Using these options can help to minimize the amount of problem determination and debugging you have to do. Other options you may find useful while debugging include:
- "-# Option" on page 91, "-v Option" on page 273, and "-V Option" on page 274
- "-qalias Option" on page 122
- "-qci Option" on page 141
- "-qobject Option" on page 207
- "-qreport Option" on page 225
- "-qsource Option" on page 239

Table 6. Options for Debugging and Error Checking

Command-Line Option	@PROCESS Directive	Description	See Page
-C -qcheck -qnocheck	CHECK NOCHECK	Checks each reference to an array element, array section, or character substring for correctness. Out-of-bounds references are reported as severe errors if found at compile time and generate **SIGTRAP** signals at run time. **Default: -qnocheck**	103

Table 6. Options for Debugging and Error Checking (continued)

Command-Line Option	@PROCESS Directive	Description	See Page
-D -qdlines -qnodlines	DLINES NODLINES	Specifies whether fixed source form lines with a D in column 1 are compiled or treated as comments. **Default: -qnodlines**	105
-g -qdbg -qnodbg	DBG NODBG	Generates debug information for use by a symbolic debugger. **Default: -qnodbg**	108
-qdpcl -qnodpcl	DPCL NODPCL	Generates symbols that tools based on the Dynamic Probe Class Library (DPCL) can use to see the structure of an executable file. **Default: -qnodpcl**	153
-qextchk -qnoextchk	EXTCHK NOEXTCHK	Sets up type-checking information for common blocks, procedure definitions, procedure references, and module data. Later, the linker can detect mismatches across compilation units by using this information. **Default: -qnoextchk**	156
-qflttrap [=*suboptions*] -qnoflttrap	FLTTRAP [(*suboptions*)] NOFLTTRAP	Determines what types of floating-point exception conditions to detect at run time. The program receives a **SIGTRAP** signal when the corresponding exception occurs. **Default: -qnoflttrap**	165
-qfullpath -qnofullpath		Records the full, or absolute, path names of source and include files in object files compiled with debugging information (-g option). **Default:** The relative path names of source files are recorded in the object files.	169
-qhalt=*sev*	HALT(*sev*)	Stops before producing any object, executable, or assembler source files if the maximum severity of compile-time messages equals or exceeds the specified severity. *severity* is one of i, l, w, e, s, u, or q, meaning informational, language, warning, error, severe error, unrecoverable error, or a severity indicating "don't stop". **Default: -qhalt=S**	170
-qinitauto[=*hex_value*] -qnoinitauto		Initializes each byte or word (4 bytes) of storage for automatic variables to a specific value, depending on the length of the *hex_value*. This helps you to locate variables that are referenced before being defined. For example, by using both the **-qinitauto** option to initialize **REAL** variables with a NaNS value and the **-qflttrap** option, it is possible to identify references to uninitialized **REAL** variables at run time. **Default: -qnoinitauto**. If you specify **-qinitauto** without a *hex_value*, the compiler initializes the value of each byte of automatic storage to zero.	177

Table 6. Options for Debugging and Error Checking (continued)

Command-Line Option	@PROCESS Directive	Description	See Page
-qlanglvl={ 77std | 90std | 90pure | 90ext | 95std | 95pure | extended}	LANGLVL({ 77STD | 90STD | 90PURE | 90EXT | 95STD | 95PURE | EXTENDED})	Determines which language standard (or superset, or subset of a standard) to consult for nonconformance. It identifies nonconforming source code and also options that allow such nonconformances. **Default: -qlanglvl=extended**	189
-qsaa -qnosaa	SAA NOSAA	Checks for conformance to the SAA FORTRAN language definition. It identifies nonconforming source code and also options that allow such nonconformances. **Default: -qnosaa**	227
-qsigtrap[= *trap_handler*]		Installs `xl__trce` or a predefined or user-written trap handler in a main program. **Default:** No trap handler installed; program core dumps when a **trap** instruction is executed.	232
-qtbtable={none | small | full}		Limits the amount of debugging traceback information in object files, to reduce the size of the program. **Default:** Full traceback information in the object file when compiling non-optimized (without **-O**) or for debugging (with **-g**). Otherwise, a small amount of traceback information in the object file.	249
-qxlines -qnoxlines	XLINES NOXLINES	Specifies whether fixed source form lines with a X in column 1 are treated as source code and compiled, or treated instead as comments. **Default: -qnoxlines**	265

Options That Control Listings and Messages

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

Some of the options in "Options for Error Checking and Debugging" on page 75 can also produce compiler messages.

Table 7. Options That Control Listings and Messages

Command-Line Option	@PROCESS Directive	Description	See Page
-#		Generates information on the progress of the compilation without actually running the individual components. **Default:** No progress messages are produced.	91

Table 7. Options That Control Listings and Messages (continued)

Command-Line Option	@PROCESS Directive	Description	See Page
-qattr[=full] -qnoattr	ATTR[(FULL)] NOATTR	Specifies whether to produce the attribute component of the attribute and cross-reference section of the listing. **Default: -qnoattr**	133
-qflag= *listing_severity*: *terminal_severity* -w	FLAG (*listing_severity*, *terminal_severity*)	Limits the diagnostic messages to those of a specified level or higher. Only messages with severity *listing_severity* or higher are written to the listing file. Only messages with severity *terminal_severity* or higher are written to the terminal. -w is a short form for -qflag=e:e. **Default: -qflag=i:i**	162
-qlist -qnolist	LIST NOLIST	Specifies whether to produce the object section of the listing. **Default: -qnolist**	195
-qlistopt -qnolistopt	LISTOPT NOLISTOPT	Determines whether to show the setting of every compiler option in the listing file or only selected options. These selected options include those specified on the command line or directives plus some that are always put in the listing. **Default: -qnolistopt**	196
-qnoprint		Prevents the listing file from being created, regardless of the settings of other listing options. **Default:** Listing is produced if you specify any of -qattr, -qlist, -qlistopt, -qphsinfo, -qreport, -qsource, or -qxref.	205
-qphsinfo -qnophsinfo	PHSINFO NOPHSINFO	Determines whether timing information is displayed on the terminal for each compiler phase. **Default: -qnophsinfo**	214
-qreport[={smplist | hotlist}...] -qnoreport	REPORT [({SMPLIST | HOTLIST}...)] NOREPORT	Determines whether to produce transformation reports showing how the program is parallelized and how loops are optimized. **Default: -qnoreport**	225
-qsource -qnosource	SOURCE NOSOURCE	Determines whether to produce the source section of the listing. **Default: -qnosource**	239
-qsuppress [= *nnnn-mmm*[:*nnnn-mmm*...] | cmpmsg] | -qnosuppress		Specifies which messages to suppress from the output stream.	245
-qxref -qnoxref -qxref=full	XREF NOXREF XREF(FULL)	Determines whether to produce the cross-reference component of the attribute and cross-reference section of the listing. **Default: -qnoxref**	267

Table 7. Options That Control Listings and Messages (continued)

Command-Line Option	@PROCESS Directive	Description	See Page
-S		Produces one or more **.s** files showing equivalent assembler source for each Fortran source file. **Default:** No equivalent assembler source is produced.	269
-v		Traces the progress of the compilation by displaying the name and parameters of each compiler component that is executed by the invocation command. **Default:** No progress messages are produced.	273
-V		Traces the progress of the compilation by displaying the name and parameters of each compiler component that is executed by the invocation command. These are displayed in a shell-executable format. **Default:** No progress messages are produced.	274

Options for Compatibility

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

Related Information: "Porting Programs to XL Fortran" on page 397 discusses this subject in more detail. "Duplicating the Floating-Point Results of Other Systems" on page 295 explains how to use some of the options in "Options for Floating-Point Processing" on page 86 to achieve floating-point results compatible with other systems.

The **-qfree=ibm** form of the "-qfree Option" on page 168 also provides compatibility with VS FORTRAN free source form.

Table 8. Options for Compatibility

Command-Line Option	@PROCESS Directive	Description	See Page
-qautodbl=*setting*	AUTODBL(*setting*)	Provides an automatic means of converting single-precision floating-point calculations to double-precision and of converting double-precision calculations to extended-precision. Use one of the following settings: none, dbl, dbl4, dbl8, dblpad, dblpad4, or dblpad8. **Default: -qautodbl=none**	134
-qcclines -qnocclines	CCLINES NOCCLINES	Determines whether the compiler recognizes conditional compilation lines. **Default: -qcclines** if you have specified the **-qsmp=omp** option; otherwise, **-qnocclines**.	139
-qctyplss [=([no]arg)] -qnoctyplss	CTYPLSS [([NO]ARG)] NOCTYPLSS	Specifies whether character constant expressions are allowed wherever typeless constants may be used. This language extension might be needed when you are porting programs from other platforms. Suboption **arg** specifies that Hollerith constants used as actual arguments will be treated as integer actual arguments. **Default: -qnoctyplss**	144
-qddim -qnoddim	DDIM NODDIM	Specifies that the bounds of pointee arrays are re-evaluated each time the arrays are referenced and removes some restrictions on the bounds expressions for pointee arrays. **Default: -qnoddim**	147

Table 8. Options for Compatibility (continued)

Command-Line Option	@PROCESS Directive	Description	See Page
-qdpc -qdpc=e -qnodpc	DPC DPC(E) NODPC	Increases the precision of real constants, for maximum accuracy when assigning real constants to **DOUBLE PRECISION** variables. This language extension might be needed when you are porting programs from other platforms. **Default: -qnodpc**	152
-qescape -qnoescape	ESCAPE NOESCAPE	Specifies how the backslash is treated in character strings, Hollerith constants, H edit descriptors, and character string edit descriptors. It can be treated as an escape character or as a backslash character. This language extension might be needed when you are porting programs from other platforms. **Default: -qescape**	154
-qextern=*names*		Allows user-written procedures to be called instead of XL Fortran intrinsics. *names* is a list of procedure names separated by colons. The procedure names are treated as if they appear in an **EXTERNAL** statement in each compilation unit being compiled. If any of your procedure names conflict with XL Fortran intrinsic procedures, use this option to call the procedures in the source code instead of the intrinsic ones. **Default:** Names of intrinsic procedures override user-written procedure names when they are the same.	157

Table 8. Options for Compatibility (continued)

Command-Line Option	@PROCESS Directive	Description	See Page
-qextname[=*name*:*name*...] -qnoextname	EXTNAME[(*name*:*name*...)] NOEXTNAME	Adds an underscore to the names of all global entities, which helps in porting programs from systems where this is a convention for mixed-language programs. **Default: -qnoextname**	158
-qinit=f90ptr	INIT(f90ptr)	Makes the initial association status of pointers disassociated. **Default:** The default association status of pointers is undefined.	176
-qintlog -qnointlog	INTLOG NOINTLOG	Specifies that you can mix integer and logical values in expressions and statements. **Default: -qnointlog**	179
-qintsize=*bytes*	INTSIZE(*bytes*)	Sets the size of default **INTEGER** and **LOGICAL** values. **Default: -qintsize=4**	180
-qlog4 -qnolog4	LOG4 NOLOG4	Specifies whether the result of a logical operation with logical operands is a **LOGICAL(4)** or is a **LOGICAL** with the maximum length of the operands. **Default: -qnolog4**	198
-qmodule=mangle81		Specifies that the compiler should use the XL Fortran Version 8.1 naming convention for non-intrinsic module files. **Default:** The compiler uses the current naming convention for non-intrinsic module names. This convention is not compatible with that used by previous versions of the compiler.	204

Table 8. Options for Compatibility (continued)

Command-Line Option	@PROCESS Directive	Description	See Page
-qnullterm -qnonullterm	NULLTERM NONULLTERM	Appends a null character to each character constant expression that is passed as a dummy argument, to make it more convenient to pass strings to C functions. **Default: -qnonullterm**	206
-1 -qonetrip -qnoonetrip	ONETRIP NOONETRIP	Executes each **DO** loop in the compiled program at least once if its **DO** statement is executed, even if the iteration count is 0. **Default: -qnoonetrip**	92
-qport [=*suboptions*] -qnoport	PORT [(*suboptions*)] NOPORT	Increases flexibility when porting programs to XL Fortran by providing a number of options to accommodate other Fortran language extensions. **Default: -qnoport**	217
-qposition= {appendold | appendunknown}	POSITION({APPENDOLD | APPENDUNKNOWN})	Positions the file pointer at the end of the file when data is written after an **OPEN** statement with no **POSITION=** specifier and the corresponding **STATUS=** value (**OLD** or **UNKNOWN**) is specified. **Default:** Depends on the I/O specifiers in the **OPEN** statement and on the compiler invocation command: **-qposition=appendold** for the **xlf**, **xlf_r**, **xlf_r7**, and **f77/fort77** commands and the defined Fortran 90 and Fortran 95 behaviors for the **xlf90**, **xlf90_r**, **xlf90_r7**, **xlf95**, **xlf95_r**, **xlf95_r7**, **f90** and **f95** commands.	219

Table 8. Options for Compatibility (continued)

Command-Line Option	@PROCESS Directive	Description	See Page
-qqcount -qnoqcount	QCOUNT NOQCOUNT	Accepts the **Q** character-count edit descriptor (**Q**) as well as the extended-precision **Q** edit descriptor (**Q***w.d*). With -qnoqcount, all **Q** edit descriptors are interpreted as the extended-precision **Q** edit descriptor. **Default: -qnoqcount**	221
-qrealsize=*bytes*	REALSIZE(*bytes*)	Sets the default size of **REAL**, **DOUBLE PRECISION**, **COMPLEX**, and **DOUBLE COMPLEX** values. **Default: -qrealsize=4**	222
-qsave[={all | defaultinit}] -qnosave	SAVE{(ALL | DEFAULTINIT)} NOSAVE	Specifies the default storage class for local variables. **-qsave**, **-qsave=all**, or **-qsave=defaultinit** sets the default storage class to **STATIC**, while **-qnosave** sets it to **AUTOMATIC**. **Default: -qnosave** **-qsave** is turned on by default for **xlf**, **xlf_r**, **xlf_r7**, **f77**, or **fort77** to duplicate the behavior of FORTRAN77 commands.	228
-qsclk=[centi | micro]		Specifies that when returning a value using the **SYSTEM_CLOCK** intrinsic procedure, the compiler will use centisecond resolution. You can specify a microsecond resolution by using **–qsclk=micro**. **Default: -qsclk=centi**	230
-qswapomp -qnoswapomp	SWAPOMP NOSWAPOMP	Specifies that the compiler should recognize and substitute OpenMP routines in XL Fortran programs. **Default: -qswapomp**	247

Table 8. Options for Compatibility (continued)

Command-Line Option	@PROCESS Directive	Description	See Page
-u -qundef -qnoundef	UNDEF NOUNDEF	Specifies whether implicit typing of variable names is permitted. -u and -qundef have the same effect as the **IMPLICIT NONE** statement that appears in each scope that allows implicit statements. **Default: -qnoundef**	272
-qwarn64 -qnowarn64		Detects the truncation of an 8-byte integer pointer to 4 bytes. Identifies, through informational messsages, statements that might cause problems during the 32-bit to 64-bit migration. **Default: -qnowarn64**	
-qxflag=oldtab	XFLAG(OLDTAB)	Interprets a tab in columns 1 to 5 as a single character (for fixed source form programs), for compatibility with XL Fortran Version 1. **Default:** Tab is interpreted as one or more characters.	259
-qxlf77=*settings*	XLF77(*settings*)	Provides backward compatibility with XL Fortran for AIX® Versions 1 and 2 aspects of language semantics and I/O data format that have changed. Most of these changes are required by the Fortran 90 standard. **Default:** Default suboptions are blankpad, nogedit77, nointarg, nointxor, leadzero, nooldboz, nopersistent, and nosofteof for the **xlf90**, **xlf90_r**, **xlf90_r7**, **xlf95**, **xlf95_r**, **xlf95_r7**, **f90**, and **f95** commands and are the exact opposite for the **xlf**, **xlf_r**, **xlf_r7**, and **f77/fort77** commands.	261

Table 8. Options for Compatibility (continued)

Command-Line Option	@PROCESS Directive	Description	See Page
-qxlf90= {[no]signedzero | [no]autodealloc}	XLF90({[no]signedzero | [no]autodealloc})	Determines whether the compiler provides the Fortran 90 or the Fortran 95 level of support for certain aspects of the language. **Default:** The default suboptions are **signedzero** and **autodealloc** for the **xlf95**, **xlf95_r**, **xlf95_r7**, and **f95** invocation commands. For all other invocation commands, the default suboptions are **nosignedzero** and **noautodealloc**.	263

Options for Floating-Point Processing

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

Related Information: See "-qflttrap Option" on page 165 and "Duplicating the Floating-Point Results of Other Systems" on page 295.

Table 9. Options for Floating-Point Processing

Command-Line Option	@PROCESS Directive	Description	See Page
-qfloat=*options*	FLOAT(*options*)	Determines how the compiler generates or optimizes code to handle particular types of floating-point calculations. **Default:** Default suboptions are **nofltint**, **fold**, **nohsflt**, **nohssngl**, **maf**, **nonans**, **norndsngl**, **norrm**, **norsqrt**, and **nostrictnmaf**; some of these settings are different with **-O3** optimization turned on or with **-qarch=ppc**.	163
-qieee={ Near | Minus | Plus | Zero} -y{n|m|p|z}	IEEE({Near | Minus | Plus | Zero})	Specifies the rounding mode for the compiler to use when evaluating constant floating-point expressions at compile time. **Default:** **-qieee=near**	175

Options That Control Linking

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

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

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

Related Information: The "-qextchk Option" on page 156 enables some extra consistency checking during linking.

Other linker options you might find helpful are the following:
- **-brename** (to change individual symbol names to avoid unresolved references)
- **-bmap** (to produce a map file showing information such as sizes of common blocks)

Table 10. Options That Control Linking

Command-Line Option	@PROCESS Directive	Description	See Page
-b64*		Instructs **ld** to bind 64-bit objects in 64-bit mode.	94
-bdynamic* -bshared* -bstatic*		These options are toggles used to control the processing of -l options and the way that shared objects are processed.	95
-bhalt:*error_level**		Specifies the maximum error level allowed before linker command processing halts. **Default:** **-bhalt:4**, as specified in the configuration file.	97
-bloadmap:*name**		Requests that a log of linker actions and messages be saved in file *name*. **Default:** No log is kept.	98
-bmaxdata:*bytes** -bmaxstack:*bytes**		Specifies the maximum amount of space to reserve for the program data segment and stack segment for programs where the size of these regions is a constraint. **Default:** Combined stack and data space is slightly less than 256 MB, or lower, depending on the limits for the user ID.	99
-brtl* -bnortl*		Determines which algorithm is used to find libraries (specified with the **-l** option).	100
-c		Produces an object file instead of an executable file. **Default:** Compile and link-edit, producing an executable file.	104
-L*dir**		Looks in the specified directory for libraries specified by the **-l** option. **Default:** **/usr/lib**	111
-l*key**		Searches the specified library file, where *key* selects the file **lib***key***.a**. **Default:** Libraries listed in **xlf.cfg**.	112
-qpic[=large | small]		Generates Position Independent Code (PIC) that can be used in shared libraries. **Default:** **-qpic=small**	216

Options That Control Other Compiler Operations

These options can help to do the following:
- Control internal size limits for the compiler

- Determine names and options for commands that are executed during compilation
- Determine the bit mode and instruction set for the target architecture

Table 11. Options That Control the Compiler Internal Operation

Command-Line Option	@PROCESS Directive	Description	See Page
-B*prefix*		Determines a substitute path name for executable files used during compilation, such as the compiler or linker. It can be used in combination with the **-t** option, which determines which of these components are affected by **-B**. **Default:** Paths for these components are defined in the configuration file, the **$PATH** environment variable, or both.	93
-F*config_file* -F*config_file*: *stanza* -F:*stanza*		Specifies an alternative configuration file, the stanza to use within the configuration file, or both. **Default:** The configuration file is **/etc/xlf.cfg**, and the stanza depends on the name of the command that executes the compiler.	107
-q32		Sets the bit mode and instruction set for a 32-bit target architecture.	281
-q64		Sets the bit mode and instruction set for a 64-bit target architecture.	282
-qlm -qnolm		Disables the license management control. **Default:** The license management control system (LM) is on by default. You must specify the compiler option **-qnolm** to disable LM.	197
-t*components*		Applies the prefix specified by the -B option to the designated components. *components* can be one or more of p, F, c, d, I, a, h, b, z, or l, with no separators, corresponding to an optimizing preprocessor, the C preprocessor, the compiler, the -S disassembler, the interprocedural analysis (IPA) tool, the assembler, the loop optimizer, the code generator, the binder, and the linker, respectively. **Default:** **-B** prefix, if any, applies to all components.	270
-W*component*,*options*		Passes the listed options to a component that is executed during compilation. *component* is p, F, c, d, I, a, z, or l, corresponding to an optimizing preprocessor, the C preprocessor, the compiler, the -S disassembler, the interprocedural analysis (IPA) tool, the assembler, the binder, and the linker, respectively. **Default:** The options passed to these programs are as follows: • Those listed in the configuration file • Any unrecognized options on the command line (passed to the linker)	275

Options That Are Obsolete or Not Recommended

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

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

Notes:

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

Table 12. Options That Are Obsolete or Not Recommended

Command-Line Option	@PROCESS Directive	Description	See Page
-qcharlen= *length*	CHARLEN (*length*)	Obsolete. It is still accepted, but it has no effect. The maximum length for character constants and subobjects of constants is 32 767 bytes (32 KB). The maximum length for character variables is 268 435 456 bytes (256 MB) in 32-bit mode. The maximum length for character variables is 2**40 bytes in 64-bit mode. These limits are always in effect and are intended to be high enough to avoid portability problems with programs that contain long strings.	
-qrecur -qnorecur	RECUR NORECUR	Not recommended. Specifies whether external subprograms may be called recursively. For new programs, use the **RECURSIVE** keyword, which provides a standard-conforming way of using recursive procedures. If you specify the -qrecur option, the compiler must assume that any procedure could be recursive. Code generation for recursive procedures may be less efficient. Using the **RECURSIVE** keyword allows you to specify exactly which procedures are recursive.	224

Detailed Descriptions of the XL Fortran Compiler Options

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

How to read the syntax information:

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

-# Option

Syntax

-#

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

Rules

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

Examining the output of this option can help you quickly and safely determine the following information for a particular compilation:
- What files are involved
- What options are in effect for each step

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

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

Related Information

The "-v Option" on page 273 and "-V Option" on page 274 produce the same output but also performs the compilation.

-1 Option

Syntax

```
-1
ONETRIP | NOONETRIP
```

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

Restrictions

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

Related Information

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

-B Option

Syntax

```
-Bprefix
```

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

Arguments

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

Rules

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

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

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

Examples

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

```
xlf95 -tchbI -B/home/jim/usr/lpp/xlf/bin/ test_suite.f
```

Once the upgrade meets the acceptance criteria, the system administrator installs it over the old level in **/usr/lpp/xlf/bin**.

Related Information

See “-t Option” on page 270, “-F Option” on page 107, “Customizing the Configuration File” on page 15, and “Running Two Levels of XL Fortran” on page 28.

-b64 Option

Syntax

```
-b64
```

The AIX operating system provides 64-bit shared object files in both **libc.a** and **libm.a**. In 64-bit mode, you can use the **-b64** linker option to instruct **ld** to bind with 64-bit objects.

Related Information

For more information on the 64-bit environment, see "Using XL Fortran in a 64-Bit Environment" on page 279. For more information on **-b64**, see *AIX General Programming Concepts*.

-bdynamic, -bshared, and -bstatic Options

Syntax

-bdynamic | -bshared | -bstatic

These options are toggles that are used to control the processing of **-l** options and the way that shared objects are processed.

The options **-bdynamic** and **-bshared** are synonymous.

When **-bstatic** is in effect, shared objects are statically linked into the output file. When **-bdynamic** is in effect, shared objects are linked dynamically.

When **-brtl** is used in conjunction with either **-bdynamic** or **-bshared**, the search for libraries specified with the **-l** option is satisfied by the suffix *.so* or *.a*. For each directory searched, a file with the suffix *.so* is looked for. If it is not found, a file with the suffix *.a* is looked for. If neither file is found, the search continues with the next directory.

Rules

These options are passed directly to the **ld** command and are not processed by XL Fortran at all.

These options are position-significant. They affect all files that are specified after the option on the command-line.

Table 13 summarizes how these options interact with **-brtl** and **-bnortl** to affect the file suffix that is being searched.

Table 13. Interaction of New Linker Options

		Position-significant	
		-bdynamic **-bshared (default)**	**-bstatic**
Global Influence	**-brtl**	.so .a	.a
	-bnortl (default)	.a	.a

Examples

```
xlf95 f.f -brtl -bshared -lmylib
```

In this case, the linker searches for the library **libmylib.so** first and then the library **libmylib.a** in each directory in the search path consecutively until either is encountered.

```
xlf95_r f.f -bdynamic -llib1 -bstatic -llib2 -brtl
```

In this case, to satisfy the first library specification, the linker searches for the library **liblib1.so** first and then the library **liblib1.a** in each directory (as described in the previous example). However, at the same time the linker only searches for **liblib2.a** in those same libraries.

Related Information

For more information on these options, see *AIX General Programming Concepts*. See also "-brtl Option" on page 100.

-bhalt Option

Syntax

```
-bhalt:error_level
```

Specifies the maximum error level that is allowed before the linker (**ld**) command halts. The default value is 4, as specified in the configuration file. If any linker command has an error return value greater than the value that is specified by the *error_level* variable, linking stops.

Rules

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

-bloadmap Option

Syntax

```
-bloadmap:name
```

Requests that a log of linker actions and messages be saved in file *name*. You can use the log to help diagnose linking problems. For example, the log contains information about type mismatches that the **-qextchk** option detected.

Rules

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

-bmaxdata, -bmaxstack Options

Syntax

```
-bmaxdata:bytes
-bmaxstack:bytes
```

Specifies the maximum amount of space to reserve for the program data segment and stack segment for programs where the size of these regions is a constraint.

Background Information

The data segment holds, among other things, heap storage that is used by the program.

If your program allocates large arrays, statically or dynamically, specify **-bmaxdata** when linking the program. The resulting executable program uses the large data model and can have a data region larger than a single segment, up to a maximum of 2 GB. Refer to the **ld** documentation in the *AIX Commands Reference* for allowable values. Note that since the compiler might create temporary arrays during compilation, it may be useful to define a value for the **-bmaxdata** compiler option in anticipation of this.

If the program has large amounts of automatic data or otherwise exceeds the soft limit on stack size for a program, specify **-bmaxstack** when you link the program. Use this option to define the soft limit up to 256 MB for 32-bit mode or up to the limit imposed by system resources for 64-bit mode. However, each main program or subprogram is limited to 256 MB per instance.

Arguments

You can specify the size as a decimal, octal (which is prefixed by **0**), or hexadecimal value (which is prefixed by **0x**).

Rules

These options are passed directly to the **ld** command and are not processed by XL Fortran at all.

Examples

```
xlf95 -O3 -qhot -bmaxdata:0x20000000 huge_data_model.f
xlf95 -O3 -qhot -bmaxstack:2000000 lots_of_automatic_data.f
```

Related Information

For a discussion of the issues involved in creating large AIX programs, see “Large Program Support Overview” in *AIX General Programming Concepts*.

-brtl Option

Syntax

```
-brtl | -bnortl
```

Determines which algorithm will be used to find libraries that are specified with the **-l** option.

Background Information

If **-brtl** is specified, run-time linking is enabled.

When used in conjunction with either **-bdynamic** or **-bshared**, the search for libraries that you specified with the **-l** option is satisfied by the suffix *.so* or *.a*. For each directory searched, a file with the suffix *.so* is looked for. If it is not found, a file with the suffix *.a* is looked for. If neither file is found, the search continues with the next directory. Table 13 on page 95 gives a graphical representation of how these options combine to affect the file suffix being searched for.

Rules

These options are passed directly to the **ld** command and are not processed by XL Fortran at all. Only the last specified of these options will be used. These options have a global effect; regardless of where they appear on the command line, they affect the entire command.

Examples

```
xlf95 -brtl f.f -lmylib
xlf95_r -bnortl f.f -bdynamic -llib1 -bstatic -llib2
```

Note that if you add **-brtl** to the end of the last example, it will override the earlier occurrence of **-bnortl**.

Related Information

For more information on these options, see *AIX General Programming Concepts*. See also “-bdynamic, -bshared, and -bstatic Options” on page 95.

-bshared Option

Related Information

See "-bdynamic, -bshared, and -bstatic Options" on page 95.

-bstatic Option

Related Information

See "-bdynamic, -bshared, and -bstatic Options" on page 95.

-C Option

Syntax

```
-C
CHECK | NOCHECK
```

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

Rules

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

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

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

Related Information

The **-C** option prevents some of the optimizations that the "-qhot Option" on page 171 performs. You may want to remove the **-C** option after debugging of your code is complete and to add the **-qhot** option to achieve a more thorough optimization.

The valid bounds for character substring expressions differ depending on the setting of the **-qzerosize** option. See "-qzerosize Option" on page 268.

"-qsigtrap Option" on page 232 and "Installing an Exception Handler" on page 298 describe how to detect and recover from **SIGTRAP** signals without ending the program.

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

-c Option

Syntax

```
-c
```

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

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

Related Information

See "-o Option" on page 116.

-D Option

Syntax

```
-D
DLINES | NODLINES
```

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

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

Related Information

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

-d Option

Syntax

```
-d
```

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

Rules

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

Related Information

See "Passing Fortran Files through the C Preprocessor" on page 40.

-F Option

Syntax

-F*config_file* | -F*config_file:stanza* | -F:*stanza*

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

The configuration file specifies different kinds of defaults, such as options for particular compilation steps and the locations of various files that the compiler requires. A default configuration file (**/etc/xlf.cfg**) is supplied at installation time. The default stanza depends on the name of the command used to invoke the compiler (**xlf90**, **xlf90_r**, **xlf90_r7**, **xlf95**, **xlf95_r**, **xlf95_r7**, **xlf**, **xlf_r**, **xlf_r7**, **f77**, or **fort77**).

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

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

Restrictions

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

Examples

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

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

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

Related Information

"Customizing the Configuration File" on page 15 explains the contents of a configuration file and tells how to select different stanzas in the file without using the **-F** option.

-g Option

Syntax

```
-g
DBG | NODBG
```

Generates debug information for use by a symbolic debugger.

Related Information

See "Debugging a Fortran 90 or Fortran 95 Program" on page 376, "A Sample dbx Session for an XL Fortran Program" on page 377, and "Symbolic Debugger Support" on page 9.

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

-I Option

Syntax

`-Idir`

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

Arguments

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

Rules

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

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

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

Related Information

The "-qmoddir Option" on page 203 puts **.mod** files in a specific directory when you compile a file that contains modules.

-k Option

Syntax

```
-k
FREE(F90)
```

Specifies that the program is in free source form.

Applicable Product Levels

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

Related Information

See "-qfree Option" on page 168 and *Free Source Form* in the *XL Fortran Enterprise Edition for AIX Language Reference*.

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

-L Option

Syntax

`-Ldir`

Looks in the specified directory for libraries that are specified by the **-l** option. If you use libraries other than the default ones in **/usr/lib**, you can specify one or more **-L** options that point to the locations of the other libraries. You can also set the **LIBPATH** environment variable, which lets you specify a search path for libraries at run time.

Rules

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

Related Information

See "Options That Control Linking" on page 86 and "Linking XL Fortran Programs" on page 42.

-l Option

Syntax

`-l`*key*

Searches the specified library file, where *key* selects the library **lib***key***.a**.

Rules

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

Related Information

See “Options That Control Linking” on page 86 and “Linking XL Fortran Programs” on page 42.

-N Option

Syntax

```
-NSbytes
SPILLSIZE(bytes)
```

Specifies the size of internal program storage areas.

Rules

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

Defaults

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

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

Related Information

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

-O Option

Syntax

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

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

Arguments

not specified
Almost all optimizations are disabled. This is equivalent to specifying **-O0** or **-qnoopt**.

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

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

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

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

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

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

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

-O4 implies the following other options:
- **-qhot**
- **-qipa**
- **-O3** (and all the options and settings that it implies)
- **-qarch=auto**
- **-qtune=auto**
- **-qcache=auto**

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

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

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

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

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

Restrictions

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

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

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

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

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

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

Related Information

"-qessl Option" on page 155 allows the use of ESSL routines.

"-qstrict Option" on page 241 shows how to turn off the effects of **-O3** that might change the semantics of a program.

"-qipa Option" on page 182, "-qhot Option" on page 171, and "-qpdf Option" on page 210 turn on additional optimizations that may improve performance for some programs.

"Optimizing XL Fortran Programs" on page 305 discusses technical details of the optimization techniques the compiler uses and some strategies you can use to get maximum performance from your code.

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

-o Option

Syntax

```
-o name
```

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

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

Defaults

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

Rules

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

Examples

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

-P Option

Syntax

```
-P{v|k}[!]
```

Invokes the selected optimizing preprocessor. Adding ! prevents the compilation step from following preprocessing.

You can specify only one of these preprocessor options on the command line:
-Pk invokes the KAP preprocessor.
-Pv invokes the VAST-2 preprocessor.

Examples

This example shows sets of preprocessor options that perform a reasonable amount of optimization:

```
xlf95 test.f -Pk -Wp,-r=3 -O  # Reasonable set of KAP options
xlf95 test.f -Pv -Wp,-ew -O   # Reasonable set of VAST-2 options
```

This example shows how to save the preprocessed output in a file so that you can see what transformations the preprocessors do:

```
# Produces KAP preprocessor output file Ploops.f
  xlf95 -Pk! -Wp,-f loops.f

# Produces VAST preprocessor output file Ploops.f
  xlf95 -Pv! -Wp,-o loops.f
```

Note: Because the preprocessors are not included as part of XL Fortran, you must purchase them separately for this example to work.

Related Information

For information about other kinds of preprocessing (for conditional compilation and macro expansion), see “Passing Fortran Files through the C Preprocessor” on page 40.

-p Option

Syntax

```
-p[g]
```

Sets up the object file for profiling.

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

-pg is like **-p**, but it produces more extensive statistics. Running a program compiled with **-pg** produces a **gmon.out** file, which you use with the **gprof** command to generate a run-time profile.

Rules

For profiling, the compiler produces monitoring code that counts the number of times each routine is called. The compiler replaces the startup routine of each subprogram with one that calls the monitor subroutine at the start. When the program ends normally, it writes the recorded information to the **mon.out** or **gmon.out** file.

Examples

```
$ xlf95 -p needs_tuning.f
$ a.out
$ prof
 .
 .
 .

profiling data
 .
 .
 .

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

detailed and verbose profiling data
 .
 .
 .
```

Related Information

For more information on profiling and the **prof** and **gprof** commands, see the *AIX Commands Reference*.

-Q Option

Syntax

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

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

Rules

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

Arguments

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

The **-Q!** option turns off inlining.

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

Restrictions

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

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

Examples

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

Related Information

See "-qipa Option" on page 182 and "Optimizing Subprogram Calls" on page 317.

-q32 Option

Related Information

See "-q32 Option" on page 281.

-q64 Option

Related Information

See "-q64 Option" on page 282.

-qalias Option

Syntax

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

Indicates whether a program contains certain categories of aliasing. The compiler limits the scope of some optimizations when there is a possibility that different names are aliases for the same storage locations. See "Optimizing XL Fortran Programs" on page 305 for information on aliasing strategies you should consider.

Arguments

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

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

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

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

Rules

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

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

Applicable Product Levels

-qalias=nostd replaces the option **-qxflag=xalias** and makes it obsolete.

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

Examples

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

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

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

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

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

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

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

        print *, a
      end program
```

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

```
! We cannot assert that this unit is free
! of array-assignment aliasing because of the assignments below.
      subroutine sub1
      integer a(10), b(10)
      equivalence (a, b(3))
      a = b          ! a and b overlap.
      a = a(10:1:-1) ! The elements of a are reversed.
      end subroutine

! When the overlapping assignment is recoded to explicitly use a
! temporary array, the array-assignment aliasing is removed.
! Although ALIAS(NOARYOVRLP) does not speed up this assignment,
! subsequent assignments of non-overlapping arrays in this unit
! are optimized.
@PROCESS ALIAS(NOARYOVRLP)
      subroutine sub2
      integer a(10), b(10), t(10)
```

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

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

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

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

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

Restrictions

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

Programs that contain nonstandard or integer **POINTER** aliasing may produce incorrect results if you do not compile them with the correct **-qalias** settings. The **xlf90**, **xlf90_r**, **xlf90_r7**, **xlf95**, **xlf95_r**, **xlf95_r7**, **f90**, and **f95** commands assume that a program contains only standard aliasing (**-qalias=aryovrlp:pteovrlp:std:nointptr**), while the **xlf_r**, **xlf_r7**, **xlf**, and **f77/fort77** commands, for compatibility with XL Fortran Version 2, assume that integer **POINTER**s may be present (**-qalias=aryovrlp:pteovrlp:std:intptr**).

-qalign Option

Syntax

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

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

Defaults

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

Arguments

[no]4K
: Specifies whether to align large data objects on page (4 KB) boundaries, for improved performance with data-striped I/O. Objects are affected depending on their representation within the object file. The affected objects are arrays and structures that are 4 KB or larger and are in static or bss storage and also CSECTs (typically **COMMON** blocks) that are 8 KB or larger. A large **COMMON** block, equivalence group containing arrays, or structure is aligned on a page boundary, so the alignment of the arrays depends on their position within the containing object. Inside a structure of non-sequence derived type, the compiler adds padding to align large arrays on page boundaries.

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

XL Fortran Option	Corresponding XL C Option
-qalign=bindc=bit_packed	**-qalign=bit_packed**
-qalign=bindc=full | power	**-qalign=full | power**
-qalign=bindc=mac68k | twobyte	**-qalign=mac68k | twobyte**
-qalign=bindc=natural	**-qalign=natural**
-qalign=bindc=packed	**-qalign=packed**

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

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

natural
: If the **natural** suboption of the **struct** option is specified, objects of

a derived type are stored with sufficient padding that components will be stored on their natural alignment boundaries, unless storage association requires otherwise. The natural alignment boundaries for objects of a type that appears in the left-hand column of the following table is shown in terms of a multiple of some number of bytes in the corresponding entry in the right-hand column of the table.

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

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

port

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

- Storage padding is the same as described above for the **natural** suboption, with the exception that the alignment of components of type complex is the same as the alignment of components of type real of the same kind.
- The padding for an object that is immediately followed by a union is inserted at the begining 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 must use **-qalign=4k** if you are using the I/O techniques that are described in "Increasing Throughput with Logical Volume I/O and Data Striping" on page 333.

Related Information

You can tell if an array has the **AUTOMATIC** attribute and is thus unaffected by **-qalign=4k** if you look for the keywords **AUTOMATIC** or **CONTROLLED AUTOMATIC** in the listing of the "-qattr Option" on page 133. This listing also shows the offsets of data objects.

-qarch Option

Syntax

`-qarch=`*`architecture`*

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

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

Arguments

The choices for *architecture* are:

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

com You can run the executable file that the compiler generated on any hardware platform supported by the compiler, because the file contains only instructions that are common to all machines. This choice is the default if you specify **-q32**.

If you specify the **-q64** and **-qarch=com** options together, the target platform is 64-bit, and the **-qarch** option is silently upgraded to **ppc64grsq**. The instruction set will be restricted to those instructions common to all 64-bit machines. See "Using XL Fortran in a 64-Bit Environment" on page 279 for details. Also, the **rndsngl** suboption of the **-qfloat** option is automatically turned on and cannot be turned off. While this yields better performance on PowerPC systems, you may get slightly different results than if you compile with **-qarch=com** and **-q32**.

pwr You can run the executable file on any POWER or POWER2 hardware platform. Because executable files for these platforms may contain instructions that are not available on PowerPC systems, they may be incompatible with those newer systems, or they may run more slowly because missing instructions are emulated through software traps.

pwr2 You can run the executable file on any POWER2 hardware platform. Because executable files for these platforms may contain instructions that are not available on POWER and PowerPC (including POWER3) systems, they may be incompatible with those systems.

Note that **pwrx** is a synonym for **pwr2**, but **pwr2** is preferable.

pwr2s
: You can run the executable file on any desktop implementation of the POWER2 Chip. This architecture belongs to the **-qarch=pwr2** group.

p2sc
: You can run the executable file on any POWER2 Super Chip hardware platform. The POWER2 Super Chip belongs to the **-qarch=pwr2** group.

601
: You can run the executable file on any PowerPC 601® hardware platform. Because the PowerPC 601 processor implements some instructions that are not present in other PowerPC implementations, programs might not run on other PowerPC processors. The **rndsngl** suboption of the **-qfloat** option is automatically turned on and cannot be turned off.

603
: You can run the executable file on any PowerPC 603® hardware platform. Because the PowerPC 603 processor implements some instructions that are not present in other PowerPC implementations, such as the optional PowerPC graphics instructions, programs might not run on other PowerPC processors. The **rndsngl** suboption of the **-qfloat** option is automatically turned on and cannot be turned off.

604
: You can run the executable file on any PowerPC 604® hardware platform. Because the PowerPC 604 processor implements some instructions that are not present in other PowerPC implementations, such as the optional PowerPC graphics instructions, programs might not run on other PowerPC processors. The **rndsngl** suboption of the **-qfloat** option is automatically turned on and cannot be turned off.

ppc
: You can run the executable file on any PowerPC hardware platform, including those that are based on the RS64I, RS64II, RS64III, 601, 603, 604, POWER3, POWER4, POWER5, PowerPC 970, and future PowerPC chips. If you specify the compiler option **-q64**, the target platform is 64-bit PowerPC, and the compiler silently upgrades the **-qarch** setting to **ppc64**. See "Using XL Fortran in a 64-Bit Environment" on page 279 for details.

  The **rndsngl** suboption of the **-qfloat** option is automatically turned on and cannot be turned off.

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

  In 64-bit mode, produces object code containing optional graphics instructions that will run on 64-bit PowerPC platforms, but not on 32-bit-only platforms, and the **-qarch** option will be silently upgraded to **-qarch=ppc64gr**.

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

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

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

rs64a
: You can run the executable file on any RS64I machine.

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

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

pwr3 You can run the executable file on any POWER3, POWER4, POWER5, or PowerPC 970 hardware platform. In previous releases, the **pwr3** setting was used to target the POWER3 and POWER4 group of processors. To have your compilation target a more general processor group, use the **ppc64grsq** setting, which includes the POWER3, POWER4, POWER5, or PowerPC 970 group of processors. Because executable files for these platforms may contain instructions that are not available on POWER, POWER2, or other PowerPC systems, they may be incompatible with those systems.

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

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

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

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

If you intend your program to run only on a particular architecture, you can use the **-qarch** option to instruct the compiler to generate code specific to that architecture. This allows the compiler to take advantage of machine-specific instructions that can improve performance. The **-qarch** option provides arguments for you to specify certain chip models; for example, you can specify **-qarch=604** to indicate that your program will be executed on PowerPC 604 hardware platforms.

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

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

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

- If your primary concern is to make a program widely distributable, keep the default (**com**). If your program is likely to be run on all types of processors equally often, do not specify any **-qarch** or **-qtune** options. The default supports only the common subset of instructions of all processors.
- If you want your program to run on more than one architecture, but to be tuned to a particular architecture, use a combination of the **-qarch** and **-qtune** options. Make sure that your **-qarch** setting covers all the processor types you intend your program to run on. If you run such a program on an unsupported processor, your program may fail at execution time.
- If the program will only be used on a single machine or can be recompiled before being used on a different machine, specify the applicable **-qarch** setting. Doing so might improve performance and is unlikely to increase compile time. If you specify the **p2sc**, **pwr2**, **pwr2s**, **rs64a**, **rs64b**, **rs64c**, **601**, **603**, **604**, **pwr3**, **pwr4**, **pwr5**, or **ppc970** suboption, you do not need to specify a separate **-qtune** option.

- If your primary concern is execution performance, you may see some speedup if you specify the appropriate **-qarch** suboption and perhaps also specify the **-qtune** and **-qcache** options. In this case, you may need to produce different versions of the executable file for different machines, which might complicate configuration management. You will need to test the performance gain to see if the additional effort is justified.
- It is usually better to target a specific architecture so your program can take advantage of the targeted machine's characteristics. For example, specifying **-qarch=pwr4** when targeting a POWER4 machine will benefit those programs that are floating-point intensive or have integer multiplies. On PowerPC systems, programs that process mainly unpromoted single-precision variables are more efficient when you specify **-qarch=ppc**. On POWER2 and POWER3 systems, programs that process mainly double-precision variables (or single-precision variables promoted to double by one of the **-qautodbl** options) become more efficient with **-qarch=pwr2**, **-qarch=pwr3**, **-qarch=pwr4** and **-qarch=pwr5**. The **-qautodbl=dblpad4** option will improve POWER and POWER2, but not POWER3, POWER4, and POWER5, which are PowerPC processors.

Other Considerations

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

Table 14. Instruction groups for PowerPC platforms

Processor	**Graphics group**	**sqrt group**	**64-bit**
601	no	no	no
603	yes	no	no
604	yes	no	no
ppc	no	no	no
ppcgr	yes	no	no
ppc64	no	no	yes
ppc64gr	yes	no	yes
ppc64grsq	yes	yes	yes
rs64a	no	no	yes
rs64b	yes	yes	yes
rs64c	yes	yes	yes
pwr3	yes	yes	yes
pwr4	yes	yes	yes
pwr5	yes	yes	yes
ppc970	yes	yes	yes

If you compile code using the **-qarch=pwr3** option, the code will run on an RS64B hardware platform but may not run on an RS64A platform because the instruction groups are not identical. Similarily, code compiled with the **-qarch=603** option will run on a POWER3 machine, but may not run on a RS64A machine.

Related Information

See "Compiling for Specific Architectures" on page 39, "-qtune Option" on page 251, and "-qcache Option" on page 137.

-qassert Option

Syntax

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

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

Arguments

nodeps Specifies that no loop-carried dependencies exist.

itercnt Specifies a value for unknown loop iteration counts.

Related Information

See "Cost Model for Loop Transformations" on page 313 for background information and instructions for using these assertions. See also the description of the **ASSERT** directive in the *XL Fortran Enterprise Edition for AIX Language Reference*.

-qattr Option

Syntax

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

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

Arguments

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

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

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

Related Information

See "Options That Control Listings and Messages" on page 77 and "Attribute and Cross-Reference Section" on page 392.

-qautodbl Option

Syntax

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

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

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

Rules

Although the POWER and POWER2 floating-point units perform **REAL(4)** calculations internally using fast **REAL(8)** arithmetic, it is often better to have these calculations done entirely using data entities that are **REAL(8)** or **DOUBLE PRECISION**. If the calculations are coded using **REAL** or **REAL(4)** data entities, the **REAL(4)**-**REAL(8)**-**REAL(4)** conversions take away the extra precision and range and also lessen performance, even though the intermediate calculations are done in IEEE double-precision.

Arguments

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

The settings you can use are as follows:

none
: Does not promote or pad any objects that share storage. This setting is the default.

dbl4
: Promotes floating-point objects that are single-precision (4 bytes in size) or that are composed of such objects (for example, **COMPLEX** or array objects):
- **REAL(4)** is promoted to **REAL(8)**.
- **COMPLEX(4)** is promoted to **COMPLEX(8)**.

This suboption requires the **libxlfpmt4.a** library during linking.

dbl8
: 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 **libxlfpmt8.a** library during linking.

dbl
: Combines the promotions that **dbl4** and **dbl8** perform.

This suboption requires the **libxlfpmt4.a** and **libxlfpmt8.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 **libxlfpmt4.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 **libxlfpmt8.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 **libxlfpmt4.a**, **libxlfpmt8.a**, and **libxlfpad.a** libraries during linking.

Rules

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

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

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

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

Restrictions

- Because these extra conversions do not apply to the PowerPC floating-point unit, this option may not produce any speedup on PowerPC machines. However, programs that you compile with it still work and gain extra precision on all machines.
- 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. To detect inconsistent **-qautodbl** settings, use the **-qextchk** option when compiling the source file.

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 413.

"-qrealsize Option" on page 222 describes another option that works like **-qautodbl**, but it only affects items that are of default kind type and does not do any padding. If you specify both the **-qrealsize** and the **-qautodbl** options, only **-qautodbl** takes effect. Also, **-qautodbl** overrides the **-qdpc** option.

"Linking 32–Bit Non-SMP Object Files Using the ld Command" on page 44 explains how to manually link additional libraries with object files that you compiled with **-qautodbl**.

-qcache Option

Syntax

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

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

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

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

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

Arguments

assoc=*number*
: Specifies the set associativity of the cache:

  0 Direct-mapped cache

  1 Fully associative cache

  n **> 1** *n*-way set-associative cache

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

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

level=*level*
: Specifies which level of cache is affected:

  1 Basic cache

2 Level-2 cache or the table lookaside buffer (TLB) if the machine has no level-2 cache

3 TLB in a machine that does have a level-2 cache

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

line=*bytes*
: Specifies the line size of the cache.

size=*Kbytes*
: Specifies the total size of this cache.

type={C|c| D|d|I|i}
: Specifies the type of cache that the settings apply to, as follows:
- **C** or **c** for a combined data and instruction cache
- **D** or **d** for the data cache
- **I** or **i** for the instruction cache

Restrictions

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

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

Examples

To tune performance for a system with a combined instruction and data level-1 cache where the cache is two-way associative, 8 KB in size, and has 64-byte cache lines:

```
xlf95 -O3 -qhot -qcache=type=c:level=1:size=8:line=64:assoc=2 file.f
```

To tune performance for a system with two levels of data cache, use two **-qcache** options:

```
xlf95 -O3 -qhot -qcache=type=D:level=1:size=256:line=256:assoc=4 \
      -qcache=type=D:level=2:size=512:line=256:assoc=2 file.f
```

To tune performance for a system with two types of cache, again use two **-qcache** options:

```
xlf95 -O3 -qhot -qcache=type=D:level=1:size=256:line=256:assoc=4 \
      -qcache=type=I:level=1:size=512:line=256:assoc=2 file.f
```

Related Information

See “-qtune Option” on page 251, “-qarch Option” on page 127, and “-qhot Option” on page 171.

-qcclines Option

Syntax

```
-qcclines | -qnocclines
CCLINES |  NOCCLINES
```

Determines whether the compiler recognizes conditional compilation lines in fixed source form and F90 free source form. IBM free source form is not supported.

Defaults

The default is **-qcclines** if the **-qsmp=omp** option is turned on; otherwise, the default is **-qnocclines**.

Related Information

See *Conditional Compilation* in the *Language Elements* section of the *XL Fortran Enterprise Edition for AIX Language Reference*.

-qcheck Option

Syntax

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

-qcheck is the long form of the "-C Option" on page 103.

-qci Option

Syntax

```
-qci=numbers
CI(numbers)
```

Specifies the identification numbers (from 1 to 255) of the **INCLUDE** lines to process. If an **INCLUDE** line has a number at the end, the file is only included if you specify that number in a **-qci** option. The set of identification numbers that is recognized is the union of all identification numbers that are specified on all occurrences of the **-qci** option.

This option allows a kind of conditional compilation because you can put code that is only sometimes needed (such as debugging **WRITE** statements, additional error-checking code, or XLF-specific code) into separate files and decide for each compilation whether to process them.

Examples

```
REAL X /1.0/
INCLUDE 'print_all_variables.f' 1
X = 2.5
INCLUDE 'print_all_variables.f' 1
INCLUDE 'test_value_of_x.f' 2
END
```

In this example, compiling without the **-qci** option simply declares **X** and assigns it a value. Compiling with **-qci=1** includes two instances of an include file, and compiling with **-qci=1:2** includes both include files.

Restrictions

Because the optional number in **INCLUDE** lines is not a widespread Fortran feature, using it may restrict the portability of a program.

Related Information

See the section on the **INCLUDE** directive in the *XL Fortran Enterprise Edition for AIX Language Reference*.

-qcompact Option

Syntax

```
-qcompact | -qnocompact
COMPACT | NOCOMPACT
```

Reduces optimizations that increase code size.

By default, some techniques the optimizer uses to improve performance, such as loop unrolling and array vectorization, may also make the program larger. For systems with limited storage, you can use **-qcompact** to reduce the expansion that takes place. If your program has many loop and array language constructs, using the **-qcompact** option will affect your application's overall performance. You may want to restrict using this option to those parts of your program where optimization gains will remain unaffected.

Rules

With **-qcompact** in effect, **-Q** and other optimization options still work; the reductions in code size come from limiting code replication that is done automatically during optimization.

-qcr Option

Syntax

`-qcr` | `-qnocr`

Allows you to control how the compiler interprets the CR (carriage return) character. By default, the CR (Hex value X'0d') or LF (Hex value X'0a') character, or the CRLF (Hex value X'0d0a') combination indicates line termination in a source file. This allows you to compile code written using a Mac OS or DOS/Windows editor.

If you specify **-qnocr**, the compiler recognizes only the LF character as a line terminator. You must specify **-qnocr** if you use the CR character for a purpose other than line termination.

-qctyplss Option

Syntax

```
-qctyplss[(=[no]arg)] | -qnoctyplss
CTYPLSS[([NO]ARG)]| NOCTYPLSS
```

Specifies whether character constant expressions are allowed wherever typeless constants may be used. This language extension might be needed when you are porting programs from other platforms.

Arguments

arg | **noarg** Suboptions retain the behavior of **-qctyplss**. Additionally, **arg** specifies that Hollerith constants used as actual arguments will be treated as integer actual arguments.

Rules

With **-qctyplss**, character constant expressions are treated as if they were Hollerith constants and thus can be used in logical and arithmetic expressions.

Restrictions

- If you specify the **-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

See *Hollerith Constants* in the *XL Fortran Enterprise Edition for AIX Language Reference* and “Passing Arguments By Reference or By Value” on page 353.

-qdbg Option

Syntax

```
-qdbg | -qnodbg
DBG | NODBG
```

-qdbg is the long form of the "-g Option" on page 108.

-qddim Option

Syntax

```
-qddim | -qnoddim
DDIM | NODDIM
```

Specifies that the bounds of pointee arrays are re-evaluated each time the arrays are referenced and removes some restrictions on the bounds expressions for pointee arrays.

Rules

By default, a pointee array can only have dimension declarators containing variable names if the array appears in a subprogram, and any variables in the dimension declarators must be dummy arguments, members of a common block, or use- or host-associated. The size of the dimension is evaluated on entry to the subprogram and remains constant during execution of the subprogram.

With the **-qddim** option:

- The bounds of a pointee array are re-evaluated each time the pointee is referenced. This process is called *dynamic dimensioning*. Because the variables in the declarators are evaluated each time the array is referenced, changing the values of the variables changes the size of the pointee array.
- The restriction on the variables that can appear in the array declarators is lifted, so ordinary local variables can be used in these expressions.
- Pointee arrays in the main program can also have variables in their array declarators.

Examples

```
@PROCESS DDIM
INTEGER PTE, N, ARRAY(10)
POINTER (P, PTE(N))
DO I=1, 10
   ARRAY(I)=I
END DO
N = 5
P = LOC(ARRAY(2))
PRINT *, PTE   ! Print elements 2 through 6.
N = 7          ! Increase the size.
PRINT *, PTE   ! Print elements 2 through 8.
END
```

-qdirective Option

Syntax

```
-qdirective[=directive_list] | -qnodirective[=directive_list]
DIRECTIVE[(directive_list)] | NODIRECTIVE[(directive_list)]
```

Specifies sequences of characters, known as trigger constants, that identify comment lines as compiler comment directives.

Background Information

A compiler comment directive is a line that is not a Fortran statement but is recognized and acted on by the compiler. To allow you maximum flexibility, any new directives that might be provided with the XL Fortran compiler in the future will be placed inside comment lines. This avoids portability problems if other compilers do not recognize the directives.

Defaults

The compiler recognizes the default trigger constant **IBM***. Specification of **-qsmp** implies **-qdirective=smp\$:\$omp:ibmp**, and, by default, the trigger constants **SMP$**, **$OMP**, and **IBMP** are also turned on. If you specify **-qsmp=omp**, the compiler ignores all trigger constants that you have specified up to that point and recognizes only the **$OMP** trigger constant. Specification of **-qthreaded** implies **-qdirective=ibmt**, and, by default, the trigger constant **IBMT** is also turned on.

Arguments

The **-qnodirective** option with no *directive_list* turns off all previously specified directive identifiers; with a *directive_list*, it turns off only the selected identifiers.

-qdirective with no *directive_list* turns on the default trigger constant **IBM*** if it has been turned off by a previous **-qnodirective**.

Notes

- Multiple **-qdirective** and **-qnodirective** options are additive; that is, you can turn directive identifiers on and off again multiple times.
- One or more *directive_list*s can be applied to a particular file or compilation unit; any comment line beginning with one of the strings in the *directive_list* is then considered to be a compiler comment directive.
- The trigger constants are not case-sensitive.
- The characters **(**, **)**, **'**, **"**, **:**, **=**, comma, and blank cannot be part of a trigger constant.
- To avoid wildcard expansion in trigger constants that you might use with these options, you can enclose them in single quotation marks on the command line. For example:

  ```
  xlf95 -qdirective='dbg*' -qnodirective='IBM*' directives.f
  ```
- This option only affects Fortran directives that the XL Fortran compiler provides, not those that any preprocessors provide.

Examples

```
! This program is written in Fortran free source form.
PROGRAM DIRECTV
INTEGER A, B, C, D, E, F
A = 1 ! Begin in free source form.
B = 2
!OLDSTYLE SOURCEFORM(FIXED)
! Switch to fixed source form for this include file.
      INCLUDE 'set_c_and_d.inc'
!IBM* SOURCEFORM(FREE)
! Switch back to free source form.
E = 5
F = 6
END
```

For this example, compile with the option **-qdirective=oldstyle** to ensure that the compiler recognizes the **SOURCEFORM** directive before the **INCLUDE** line. After processing the include-file line, the program reverts back to free source form, after the **SOURCEFORM(FREE)** statement.

Related Information

See the section on the **SOURCEFORM** directive in the *XL Fortran Enterprise Edition for AIX Language Reference*.

As the use of incorrect trigger constants can generate warning messages or error messages or both, you should check the particular directive statement in the *Directives* section of the *XL Fortran Enterprise Edition for AIX Language Reference* for the suitable associated trigger constant.

-qdirectstorage Option

Syntax

`-qdirectstorage` | **`-qnodirectstorage`**

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

Use this option with discretion. It is intended for programmers who know how the memory and cache blocks work, and how to tune their applications for optimal performance. For a program to execute correctly on all PowerPC implementations of cache organization, the programmer should assume that separate instruction and data caches exist, and should program to the separate cache model.

Note: Using the **-qdirectstorage** option together with the **CACHE_ZERO** directive may cause your program to fail, or to produce incorrect results..

-qdlines Option

Syntax

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

-qdlines is the long form of the "-D Option" on page 105.

-qdpc Option

Syntax

```
-qdpc[=e] | -qnodpc
DPC[(E)] | NODPC
```

Increases the precision of real constants, for maximum accuracy when assigning real constants to **DOUBLE PRECISION** variables. This language extension might be needed when you are porting programs from other platforms.

Rules

If you specify **-qdpc**, 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 **-qdpc=e**, all single-precision constants, including constants with an `e` exponent, are treated as double-precision constants.

This option does not affect constants with a kind type parameter specified.

Examples

```
@process nodpc
        subroutine nodpc
        real x
        double precision y
        data x /1.000000000001/  ! The trailing digit is lost
        data y /1.000000000001/  ! The trailing digit is lost

        print *, x, y, x .eq. y  ! So x is considered equal to y
        end

@process dpc
        subroutine dpc
        real x
        double precision y
        data x /1.000000000001/  ! The trailing digit is lost
        data y /1.000000000001/  ! The trailing digit is preserved

        print *, x, y, x .eq. y  ! So x and y are considered different
        end

        program testdpc
        call nodpc
        call dpc
        end
```

When compiled, this program prints the following:

```
   1.000000000       1.00000000000000000     T
   1.000000000       1.00000000000100009     F
```

showing that with **-qdpc** the extra precision is preserved.

Related Information

"-qautodbl Option" on page 134 and "-qrealsize Option" on page 222 are more general-purpose options that can also do what **-qdpc** does. **-qdpc** has no effect if you specify either of these options.

-qdpcl Option

Syntax

`-qdpcl | -qnodpcl`
`DPCL | NODPCL`

Generates symbols that tools based on the Dynamic Probe Class Library (DPCL) can use to see the structure of an executable file.

When you specify the **-qdpcl** option, the compiler emits symbols to define blocks of code in a program. You can then use tools that use the DPCL interface to examine performance information, such as memory usage, for object files that you compiled with this option.

Restrictions

You must also specify the **-g** option when you specify **-qdpcl**.

-qescape Option

Syntax

```
-qescape | -qnoescape
ESCAPE | NOESCAPE
```

Specifies how the backslash is treated in character strings, Hollerith constants, H edit descriptors, and character string edit descriptors. It can be treated as an escape character or as a backslash character. This language extension might be needed when you are porting programs from other platforms.

Defaults

By default, the backslash is interpreted as an escape character in these contexts. If you specify **-qnoescape**, the backslash is treated as the backslash character.

The default setting is useful for the following:

- Porting code from another Fortran compiler that uses the backslash as an escape character.
- Including "unusual" characters, such as tabs or newlines, in character data. Without this option, the alternative is to encode the ASCII values (or EBCDIC values, on mainframe systems) directly in the program, making it harder to port.

If you are writing or porting code that depends on backslash characters being passed through unchanged, specify **-qnoescape** so that they do not get any special interpretation. You could also write `\\` to mean a single backslash character under the default setting.

Examples

```
$ # Demonstrate how backslashes can affect the output
$ cat escape.f
       PRINT *,'a\bcde\fg'
       END
$ xlf95 escape.f
** _main === End of Compilation 1 ===
1501-510  Compilation successful for file escape.f.
$ a.out
cde
    g
$ xlf95 -qnoescape escape.f
** _main === End of Compilation 1 ===
1501-510  Compilation successful for file escape.f.
 $ a.out
  a\bcde\fg
```

In the first compilation, with the default setting of **-qescape**, **\b** is printed as a backspace, and **\f** is printed as a formfeed character. With the **-qnoescape** option specified, the backslashes are printed like any other character.

Related Information

The list of escape sequences that XL Fortran recognizes is shown in Table 25 on page 351.

-qessl Option

Syntax

`-qessl` | **`-qnoessl`**

Allows the use of ESSL routines in place of Fortran 90 intrinsic procedures.

The Engineering and Scientific Subroutine Library (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 the RS/6000 workstations. Some of the Fortran 90 intrinsic procedures have similar counterparts in ESSL. Performance is improved when these Fortran 90 intrinsic procedures are linked with ESSL. In this case, you can keep the interface of Fortran 90 intrinsic procedures, and get the added benefit of improved performance using ESSL.

Rules

Use the ESSL Serial Library when linking with **-lessl**. Use the ESSL SMP Library when linking with **-lesslsmp**.

Either -lessl or -lesslsmp must be used whenever code is being compiled with **-qessl**. ESSL v3.1.2 or above is recommended. It supports both 32-bit and 64-bit environments.

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)
```

Examples

Related Information

The ESSL libraries are not shipped with the XL Fortran compiler. For more information on these two libraries, see the *Engineering and Scientific Subroutine Library for AIX Guide and Reference*.

-qextchk Option

Syntax

```
-qextchk | -qnoextchk
EXTCHK | NOEXTCHK
```

Sets up type-checking information for common blocks, procedure definitions, procedure references, and module data. Later, the linker can detect mismatches across compilation units by using this information.

Rules

At compile time, **-qextchk** verifies the consistency of procedure definitions and references and module data.

At link time, **-qextchk** verifies that actual arguments agree in type, shape, passing mode, and class with the corresponding dummy arguments and that declarations of common blocks and modules are consistent.

If null arguments are used in a procedure reference, the compiler will not verify that the actual arguments agree with the corresponding dummy arguments at both compile and link time.

-qextern Option

Syntax

```
-qextern=names
```

Allows user-written procedures to be called instead of XL Fortran intrinsics. *names* is a list of procedure names separated by colons. The procedure names are treated as if they appear in an **EXTERNAL** statement in each compilation unit being compiled. If any of your procedure names conflict with XL Fortran intrinsic procedures, use this option to call the procedures in the source code instead of the intrinsic ones.

Arguments

Separate the procedure names with colons.

Applicable Product Levels

Because of the many Fortran 90 and Fortran 95 intrinsic functions and subroutines, you might need to use this option even if you did not need it for FORTRAN 77 programs.

Examples

```
SUBROUTINE GETENV(VAR)
   CHARACTER(10) VAR
   PRINT *,VAR
END

CALL GETENV('USER')
END
```

Compiling this program with no options fails because the call to **GETENV** is actually calling the intrinsic subroutine, not the subroutine defined in the program. Compiling with **-qextern=getenv** allows the program to be compiled and run successfully.

-qextname Option

Syntax

```
-qextname[=name1[:name2...]] | -qnoextname
EXTNAME[(name1: name2:...)] | NOEXTNAME
```

Adds an underscore to the names of all global entities, which helps in porting programs from systems where this is a convention for mixed-language programs. Use **-qextname=name1[:name2...]** to identify a specific global entity (or entities). For a list of named entities, separate each name with a colon.

The name of a main program is not affected.

The **-qextname** option helps to port mixed-language programs to XL Fortran without modifications. Use of this option avoids naming problems that might otherwise be caused by:

- Fortran subroutines, functions, or common blocks that are named **main**, **MAIN**, or have the same name as a system subroutine
- Non-Fortran routines that are referenced from Fortran and contain an underscore at the end of the routine name

  Note: XL Fortran Service and Utility Procedures, such as **flush_** and **dtime_**, have these underscores in their names already. By compiling with the **-qextname** option, you can code the names of these procedures without the trailing underscores.
- Non-Fortran routines that call Fortran procedures and use underscores at the end of the Fortran names
- Non-Fortran external or global data objects that contain an underscore at the end of the data name and are shared with a Fortran procedure

If your program has only a few instances of the naming problems that **-qextname** solves, you may prefer to select new names with the **-brename** option of the **ld** command.

Restrictions

You must compile all the source files for a program, including the source files of any required module files, with the same **-qextname** setting.

If you use the **xlfutility** module to ensure that the Service and Utility subprograms are correctly declared, you must change the name to **xlfutility_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 **xlfutility_extname** module may cause the procedure declaration check not to work accurately.

Examples

```
@PROCESS EXTNAME
        SUBROUTINE STORE_DATA
        CALL FLUSH(10)  ! Using EXTNAME, we can drop the final underscore.
        END SUBROUTINE
```

```
@PROCESS(EXTNAME(sub1))
program main
  external :: sub1, sub2
  call sub1()              ! An underscore is added.
  call sub2()              ! No underscore is added.
end program
```

Related Information

This option also affects the names that are specified in several other options, so you do not have to include underscores in their names on the command line. The affected options are "-qextern Option" on page 157, "-Q Option" on page 119, and "-qsigtrap Option" on page 232.

-qfdpr Option

Syntax

`-qfdpr | -qnofdpr`

Provides object files with information that the AIX Feedback Directed Program Restructuring (fdpr) performance-tuning utility needs to optimize the resulting executable file.

Restrictions

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.

If you use the **-qfdpr** compiler option, only those object files that are built with this flag will be reordered. Therefore, if you use **-qfdpr**, you should use it for all object files in a program. Static linking will not improve performance if you use the **-qfdpr** compiler option.

When you use **-qfdpr** on some of the objects that are built into an executable, fdpr will only perform some optimizations on the objects that are built with fdpr. This can mean that fdpr has less benefit on programs compiled using **-qfdpr**, because library code is not optimized (since it has not been compiled with **-qfdpr**).

The optimizations that the fdpr command performs are similar to those that the **-qpdf** option performs.

Related Information

For more information, see the fdpr man page and the *AIX Commands Reference*.

-qfixed Option

Syntax

```
-qfixed[=right_margin]
FIXED[(right_margin)]
```

Indicates that the input source program is in fixed source form and optionally specifies the maximum line length.

The source form specified when executing the compiler applies to all of the input files, although you can switch the form for a compilation unit by using a **FREE** or **FIXED @PROCESS** directive or switch the form for the rest of the file by using a **SOURCEFORM** comment directive (even inside a compilation unit).

For source code from some other systems, you may find you need to specify a right margin larger than the default. This option allows a maximum right margin of 132.

Defaults

-qfixed=72 is the default for the **xlf**, **xlf_r**, **xlf_r7**, **f77**, and **fort77** commands.
-qfree=f90 is the default for the **f90**, **f95**, **xlf90**, **xlf90_r**, **xlf90_r7**, **xlf95**, **xlf95_r**, and **xlf95_r7** commands.

Related Information

See “-qfree Option” on page 168.

For the precise specifications of this source form, see *Fixed Source Form* in the *XL Fortran Enterprise Edition for AIX Language Reference*.

-qflag Option

Syntax

```
-qflag=listing_severity:terminal_severity
FLAG(listing_severity,terminal_severity)
```

You must specify both *listing_severity* and *terminal_severity*.

Limits the diagnostic messages to those of a specified level or higher. Only messages with severity *listing_severity* or higher are written to the listing file. Only messages with severity *terminal_severity* or higher are written to the terminal. **-w** is a short form for **-qflag=e:e**.

Arguments

The severity levels (from lowest to highest) are:

i Informational messages. They explain things that you should know, but they usually do not require any action on your part.

l Language-level messages, such as those produced under the **-qlanglvl** option. They indicate possible nonportable language constructs.

w Warning messages. They indicate error conditions that might require action on your part, but the program is still correct.

e Error messages. They indicate error conditions that require action on your part to make the program correct, but the resulting program can probably still be executed.

s Severe error messages. They indicate error conditions that require action on your part to make the program correct, and the resulting program will fail if it reaches the location of the error. You must change the **-qhalt** setting to make the compiler produce an object file when it encounters this kind of error.

u Unrecoverable error messages. They indicate error conditions that prevent the compiler from continuing. They require action on your part before you can compile your program.

q No messages. A severity level that can never be generated by any defined error condition. Specifying it prevents the compiler from displaying messages, even if it encounters unrecoverable errors.

The **-qflag** option overrides any **-qlanglvl** or **-qsaa** options specified.

Defaults

The default for this option is **i:i** so that you do not miss any important informational messages.

Related Information

See “-qlanglvl Option” on page 189 and “Understanding XL Fortran Error Messages” on page 369.

-qfloat Option

Syntax

```
-qfloat=options
FLOAT(options)
```

Selects different strategies for speeding up or improving the accuracy of floating-point calculations.

This option replaces several separate options. For any new code, you should use it instead of **-qfold**, **-qmaf**, or related options.

You should be familiar with the information in "XL Fortran Floating-Point Processing" on page 287 and the IEEE standard before attempting to change any **-qfloat** settings.

Defaults

The default setting uses the suboptions **nofltint**, **fold**, **nohsflt**, **nohssngl**, **maf**, **nonans**, **norndsngl**, **norrm**, **norsqrt**, and **nostrictnmaf**. Some options change this default, as explained below.

The default setting of each suboption remains in effect unless you explicitly change it. For example, if you select **-qfloat=nofold**, the settings for **nohsflt**, **nohssngl**, or related options are not affected.

Arguments

The available suboptions each have a positive and negative form, such as **fold** and **nofold**, where the negative form is the opposite of the positive.

The suboptions are as follows:

fltint | <u>**nofltint**</u>

Speeds up floating-point-to-integer conversions by using an inline sequence of code instead of a call to a library function.

The library function, which is called by default if **-qfloat=fltint** is not specified or implied by another option, checks for floating-point values outside the representable range of integers and returns the minimum or maximum representable integer if passed an out-of-range floating-point value.

The Fortran language does not require checking for floating-point values outside the representable range of integers. In order to improve efficiency, the inline sequence used by **-qfloat=fltint** does not perform this check. If passed a value that is out of range, the inline sequence will produce undefined results.

Although this suboption is turned off by default, it is turned on by the **-O3** optimization level unless you also specify **-qstrict**.

<u>**fold**</u> | **nofold**

Evaluates constant floating-point expressions at compile time, which may yield slightly different results from evaluating them at run time. The compiler always evaluates constant expressions in specification statements, even if you specify **nofold**.

hsflt | <u>**nohsflt**</u>

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.

Notes:

1. This suboption is intended for specific applications in which floating-point calculations have known characteristics. In particular, all floating-point results must be within the defined range of representation of single precision. The use of this option when compiling other application programs may produce incorrect results without warning. See "Technical Details of the -qfloat=hsflt Option" on page 412 for details.

hssngl | nohssngl

Speeds up calculations in a safer way than **hsflt**, by rounding single-precision expressions only when the results are stored into **REAL(4)** memory locations.

maf | nomaf

Makes floating-point calculations faster and more accurate by using multiply-add instructions where appropriate. The possible disadvantage is that results may not be exactly equivalent to those from similar calculations that are performed at compile time or on other types of computers. Also, negative zero may be produced.

nans | nonans

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

rndsngl | norndsngl

Rounds the result of each single-precision (**REAL(4)**) operation to single-precision, rather than waiting until the full expression is evaluated. It sacrifices speed for consistency with results from similar calculations on other types of computers.

This setting is always in effect for programs that you compile with any of the **-qarch** PowerPC suboptions , because of the way the PowerPC floating-point unit works. The **rndsngl** suboption is also turned on if you specify **-q64** and **-qarch=com** together.

rrm | norrm

Turns off compiler optimizations that require the rounding mode to be the default, round-to-nearest, at run time. Use this option if your program changes the rounding mode by any means, such as by calling the **fpsets** procedure. Otherwise, the program may compute incorrect results.

rsqrt | norsqrt

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

Although this suboption is turned off by default, specifying **-O3** turns it on unless you also specify **-qstrict**.

strictnmaf | nostrictnmaf

Turns off floating-point transformations that are used to introduce negative MAF instructions, as these transformations do not preserve the sign of a zero value. By default, the compiler enables these types of transformations.

To ensure strict semantics, specify both **-qstrict** and **-qfloat=strictnmaf**.

-qflttrap Option

Syntax

```
-qflttrap[=suboptions] | -qnoflttrap
FLTTRAP[(suboptions)] | NOFLTTRAP
```

Determines what types of floating-point exception conditions to detect at run time. The program receives a **SIGTRAP** signal when the corresponding exception occurs.

Arguments

ENable
: Turn on checking for the specified exceptions in the main program so that the exceptions generate **SIGTRAP** signals. You must specify this suboption if you want to turn on exception trapping without modifying your source code.

IMPrecise
: Only check for the specified exceptions on subprogram entry and exit. This suboption improves performance, but it can make the exact spot of the exception difficult to find.

INEXact
: Detect and trap on floating-point inexact if exception-checking is enabled. Because inexact results are very common in floating-point calculations, you usually should not need to turn this type of exception on.

INValid
: Detect and trap on floating-point invalid operations if exception-checking is enabled.

NANQ
: Detect and trap all quiet not-a-number values (NaNQs) and signaling not-a-number values (NaNSs). Trapping code is generated regardless of specifying the **enable** or **imprecise** suboption. This suboption detects all NaN values handled by or generated by floating point instructions, including those not created by invalid operations. This option can impact performance.

OVerflow
: Detect and trap on floating-point overflow if exception-checking is enabled.

UNDerflow
: Detect and trap on floating-point underflow if exception-checking is enabled.

ZEROdivide
: Detect and trap on floating-point division by zero if exception-checking is enabled.

Defaults

The **-qflttrap** option without suboptions is equivalent to **-qflttrap=ov:und:zero:inv:inex**. However, because this default does not include **enable**, it is probably only useful if you already use **fpsets** or similar subroutines in your source. If you specify **-qflttrap** more than once, both with and without suboptions, the **-qflttrap** without suboptions is ignored.

Restrictions

On AIX Version 5.1 and above, if you use **-qflttrap=inv:en** to compile a program containing an IEEE invalid **SQRT** operation and then run that program, the expected **SIGTRAP** signal may not occur on PowerPC machines and does not occur at all on POWER machines.

You can only fix this problem for AIX Version 5.1 and subsequent levels of the operating system. Specify the following command:

```
export SQRT_EXCEPTION=3.1
```

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 -qflttrap=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 **SIGTRAP** signal when the exception occurs. The **-qsigtrap** option produces informative output when the signal stops the program.

Related Information

See “-qsigtrap Option” on page 232.

See “Detecting and Trapping Floating-Point Exceptions” on page 296 for full instructions on how and when to use the **-qflttrap** option, especially if you are just starting to use it.

-qfree Option

Syntax

```
-qfree[={f90|ibm}]
FREE[({F90|IBM})]
```

Indicates that the source code is in free source form. The **ibm** and **f90** suboptions specify compatibility with the free source form defined for VS FORTRAN and Fortran 90, respectively. Note that the free source form defined for Fortran 90 also applies to Fortran 95.

The source form specified when executing the compiler applies to all of the input files, although you can switch the form for a compilation unit by using a **FREE** or **FIXED @PROCESS** directive or for the rest of the file by using a **SOURCEFORM** comment directive (even inside a compilation unit).

Defaults

-qfree by itself specifies Fortran 90 free source form.

-qfixed=72 is the default for the **xlf**, **xlf_r**, **xlf_r7**, **f77**, and **fort77** commands. **-qfree=f90** is the default for the **f90**, **f95**, **xlf90**, **xlf90_r**, **xlf90_r7**, **xlf95**, **xlf95_r**, and **xlf95_r7** commands.

Related Information

See "-qfixed Option" on page 161.

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

Fortran 90 free source form is explained in *Free Source Form* in the *XL Fortran Enterprise Edition for AIX Language Reference*. It is the format to use for maximum portability across compilers that support Fortran 90 and Fortran 95 features now and in the future.

IBM free source form is equivalent to the free format of the IBM VS FORTRAN compiler, and it is intended to help port programs from the z/OS® platform. It is explained in *IBM Free Source Form* in the *XL Fortran Enterprise Edition for AIX Language Reference*.

-qfullpath Option

Syntax

```
-qfullpath | -qnofullpath
```

Records the full, or absolute, path names of source and include files in object files compiled with debugging information (**-g** option).

If you need to move an executable file into a different directory before debugging it or have multiple versions of the source files and want to ensure that the debugger uses the original source files, use the **-qfullpath** option in combination with the **-g** option so that source-level debuggers can locate the correct source files.

Defaults

By default, the compiler records the relative path names of the original source file in each **.o** file. It may also record relative path names for include files.

Restrictions

Although **-qfullpath** works without the **-g** option, you cannot do source-level debugging unless you also specify the **-g** option.

Examples

In this example, the executable file is moved after being created, but the debugger can still locate the original source files:

```
$ xlf95 -g -qfullpath file1.f file2.f file3.f -o debug_version
...
$ mv debug_version $HOME/test_bucket
$ cd $HOME/test_bucket
$ dbx debug_version
```

Related Information

See "-g Option" on page 108.

-qhalt Option

Syntax

```
-qhalt=severity
HALT(severity)
```

Stops before producing any object, executable, or assembler source files if the maximum severity of compile-time messages equals or exceeds the specified severity. *severity* is one of **i**, **l**, **w**, **e**, **s**, **u**, or **q**, meaning informational, language, warning, error, severe error, unrecoverable error, or a severity indicating "don't stop".

Arguments

The severity levels (from lowest to highest) are:

i Informational messages. They explain things that you should know, but they usually do not require any action on your part.

l Language-level messages, such as those produced under the **-qlanglvl** option. They indicate possible nonportable language constructs.

w Warning messages. They indicate error conditions that might require action on your part, but the program is still correct.

e Error messages. They indicate error conditions that require action on your part to make the program correct, but the resulting program can probably still be executed.

s Severe error messages. They indicate error conditions that require action on your part to make the program correct, and the resulting program will fail if it reaches the location of the error. You must change the **-qhalt** setting to make the compiler produce an object file when it encounters this kind of error.

u Unrecoverable error messages. They indicate error conditions that prevent the compiler from continuing. They require action on your part before you can compile your program.

q No messages. A severity level that can never be generated by any defined error condition. Specifying it prevents the compiler from displaying messages, even if it encounters unrecoverable errors.

Defaults

The default is **-qhalt=s**, which prevents the compiler from generating an object file when compilation fails.

Restrictions

The **-qhalt** option can override the **-qobject** option, and **-qnoobject** can override **-qhalt**.

-qhot Option

Syntax

```
-qhot[=suboptions] | -qnohot
HOT[=suboptions] | NOHOT
```

The **-qhot** compiler option is a powerful alternative to hand tuning that provides opportunities to optimize loops and array language. The **-qhot** compiler option will always attempt to optimize loops, regardless of the suboptions you specify.

If you do not specify an optimization level of **2** or higher when using **-O** and **-qhot**, the compiler assumes **-O2**.

For additional information on loop unrolling, see the Directives for Loop Optimization section in the *XL Fortran Enterprise Edition for AIX Language Reference*.

Array Padding: In XL Fortran, array dimensions that are powers of two can lead to a decrease in cache utilization. The **arraypad** suboption allows the compiler to increase the dimensions of arrays where doing so could improve the efficiency of array-processing loops. This can reduce cache misses and page faults that slow your array processing programs.

Specify **-qhot=arraypad** when your source includes large arrays with dimensions that are powers of 2. This can be particularly effective when the first dimension is a power of 2.

The **-C** option turns off some array optimizations.

Vectorization: The **-qhot** compiler option supports the **vector** suboption that can optimize loops in source code for operations on array data, by ensuring that operations run in parallel where applicable. The compiler uses standard registers with no vector size restrictions. Supporting single and double-precision floating-point mathematics, users can typically see benefit when applying **–qhot=vector** to applications with significant mathematical requirements.

Arguments

arraypad
: The compiler pads any arrays where there could be an increase in cache utilization. Not all arrays will necessarily be padded, and the compiler can pad different arrays by different amounts.

arraypad=*n*
: The compiler pads each array in the source. The pad amount must be a positive integer value. Each array will be padded by an integral number of elements. The integral value *n* must be multiples of the largest array element size for effective use of **arraypad**. This value is typically 4, 8, or 16.

When you specify the **arraypad** and **arraypad=n** options, the compiler does not check for reshaping or equivalences. If padding takes place, your program can produce unexpected results.

vector | **novector**
: The compiler converts certain operations in a loop that apply to successive elements of an array into a call to a routine that is in the **libxlopt.a** library. This call calculates several results at one time, which is faster than calculating each result sequentially.

If you specify **-qhot=novector**, the compiler performs optimizations on loops and arrays, but avoids replacing certain code with calls to vector library routines. The **-qhot=vector** option can affect the precision of your program's results so you should specify either **-qhot=novector** or **-qstrict** if the change in precision is unacceptable to you.

Defaults

- The **-qhot=vector** suboption is on by default when you specify the **-qhot**, **-qsmp**, **-O4**, or **-O5** options.

Examples

The following example turns on the **-qhot=vector** option but then turns it off before the compiler processes the code.

```
xlf95 -qhot=vector t.f -qhot=novector
```

Related Information

"Optimizing Loops and Array Language" on page 312 lists the transformations that are performed.

-qhsflt Option

Syntax

```
-qhsflt | -qnohsflt
HSFLT | NOHSFLT
```

Obsolete. Replaced by the **hsflt** and **nohsflt** suboptions of the "-qfloat Option" on page 163.

Related Information

Be sure to read "Maximizing Floating-Point Performance" on page 295 and "Technical Details of the -qfloat=hsflt Option" on page 412 for information about the intended purpose and restrictions of this option.

-qhssngl Option

Syntax

```
-qhssngl | -qnohssngl
HSSNGL | NOHSSNGL
```

Obsolete. Replaced by the **hssngl** and **nohssngl** suboptions of the “-qfloat Option” on page 163.

-qieee Option

Syntax

```
-qieee={Near | Minus | Plus | Zero}
IEEE({Near | Minus | Plus | Zero})
```

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

Arguments

The choices are:

Near Round to nearest representable number.

Minus Round toward minus infinity.

Plus Round toward plus infinity.

Zero Round toward zero.

This option is intended for use in combination with the XL Fortran subroutine **fpsets** or some other method of changing the rounding mode at run time. It sets the rounding mode that is used for compile-time arithmetic (for example, evaluating constant expressions such as **2.0/3.5**). By specifying the same rounding mode for compile-time and run-time operations, you can avoid inconsistencies in floating-point results.

Note: Compile-time arithmetic is most extensive when you also specify the **-O** option.

If you change the rounding mode to other than the default (round-to-nearest) at run time, be sure to also specify **-qfloat=rrm** to turn off optimizations that only apply in the default rounding mode.

Related Information

See "Selecting the Rounding Mode" on page 292, "-O Option" on page 114, and "-qfloat Option" on page 163.

-qinit Option

Syntax

```
-qinit=f90ptr
INIT(F90PTR)
```

Makes the initial association status of pointers disassociated. Note that this applies to Fortran 95 as well as Fortran 90, and above.

You can use this option to help locate and fix problems that are due to using a pointer before you define it.

Related Information

See *Pointer Association* in the *XL Fortran Enterprise Edition for AIX Language Reference*.

-qinitauto Option

Syntax

`-qinitauto[=hex_value] | -qnoinitauto`

Initializes each byte or word (4 bytes) of storage for automatic variables to a specific value, depending on the length of the *hex_value*. This helps you to locate variables that are referenced before being defined. For example, by using both the **-qinitauto** option to initialize **REAL** variables with a NaNS value and the **-qflttrap** option, it is possible to identify references to uninitialized **REAL** variables at run time. Prior to XL Fortran Version 5.1.1, you could only use this option to initialize each byte of storage.

Setting *hex_value* to zero ensures that all automatic variables are cleared before being used. Some programs assume that variables are initialized to zero and do not work when they are not. Other programs may work if they are not optimized but fail when they are optimized. Typically, setting all the variables to all zero bytes prevents such run-time errors. It is better to locate the variables that require zeroing and insert code in your program to do so than to rely on this option to do it for you. Using this option will generally zero more things than necessary and may result in slower programs.

To locate and fix these errors, set the bytes to a value other than zero, which will consistently reproduce incorrect results. This method is especially valuable in cases where adding debugging statements or loading the program into a symbolic debugger makes the error go away.

Setting *hex_value* to **FF** (255) gives **REAL** and **COMPLEX** variables an initial value of "negative not a number", or -NaNQ. Any operations on these variables will also result in NaNQ 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 NaNS value. You can initialize 8-byte **REAL** variables to double-precision NaNS values by specifying an 8-digit hexadecimal number, that, when repeated, has a double-precision NaNS value. For example, you could specify a number such as 7FBFFFFF, that, when stored in a **REAL(4)** variable, has a single-precision NaNS value. The value 7FF7FFFF, when stored in a **REAL(4)** variable, has a single-precision NaNQ value. If the same number is stored twice in a **REAL(8)** variable (7FF7FFFF7FF7FFFF), it has a double-precision NaNS value.

Arguments

- The *hex_value* is a 1-digit to 8-digit hexadecimal (0-F) number.
- To initialize each byte of storage to a specific value, specify 1 or 2 digits for the *hex_value*. If you specify only 1 digit, the compiler pads the *hex_value* on the left with a zero.
- To initialize each word of storage to a specific value, specify 3 to 8 digits for the *hex_value*. If you specify more than 2 but fewer than 8 digits, the compiler pads the *hex_value* on the left with zeros.
- In the case of word initialization, if automatic variables are not a multiple of 4 bytes in length, the *hex_value* may be truncated on the left to fit. For example, if you specify 5 digits for the *hex_value* and an automatic variable is only 1 byte long, the compiler truncates the 3 digits on the left-hand side of the *hex_value* and assigns the two right-hand digits to the variable.

- You can specify alphabetic digits as either upper- or lower-case.

Defaults

- By default, the compiler does not initialize automatic storage to any particular value. However, it is possible that a region of storage contains all zeros.
- If you do not specify a *hex_value* suboption for **-qinitauto**, the compiler initializes the value of each byte of automatic storage to zero.

Restrictions

- Equivalenced variables, structure components, and array elements are not initialized individually. Instead, the entire storage sequence is initialized collectively.

Examples

The following example shows how to perform word initialization of automatic variables:

```
subroutine sub()
integer(4), automatic :: i4
character, automatic :: c
real(4), automatic :: r4
real(8), automatic :: r8
end subroutine
```

When you compile the code with the following option, the compiler performs word initialization, as the *hex_value* is longer than 2 digits:

```
-qinitauto=0cf
```

The compiler initializes the variables as follows, padding the *hex_value* with zeros in the cases of the `i4`, `r4`, and `r8` variables and truncating the first hexadecimal digit in the case of the `c` variable:

Variable	Value
i4	000000CF
c	CF
r4	000000CF
r8	000000CF000000CF

Related Information

See "-qflttrap Option" on page 165 and the section on the **AUTOMATIC** directive in the *XL Fortran Enterprise Edition for AIX Language Reference*.

-qintlog Option

Syntax

```
-qintlog | -qnointlog
INTLOG | NOINTLOG
```

Specifies that you can mix integer and logical data entities in expressions and statements. Logical operators that you specify with integer operands act upon those integers in a bit-wise manner, and integer operators treat the contents of logical operands as integers.

Restrictions

The following operations do not allow the use of logical variables:
- **ASSIGN** statement variables
- Assigned **GOTO** variables
- **DO** loop index variables
- Implied-**DO** loop index variables in **DATA** statements
- Implied-**DO** loop index variables in either input and output or array constructors
- Index variables in **FORALL** constructs

Examples

```
INTEGER I, MASK, LOW_ORDER_BYTE, TWOS_COMPLEMENT
I = 32767
MASK = 255
! Find the low-order byte of an integer.
LOW_ORDER_BYTE = I .AND. MASK
! Find the twos complement of an integer.
TWOS_COMPLEMENT = .NOT. I
END
```

Related Information

You can also use the intrinsic functions **IAND**, **IOR**, **IEOR**, and **NOT** to perform bitwise logical operations.

-qintsize Option

Syntax

```
-qintsize=bytes
INTSIZE(bytes)
```

Sets the size of default **INTEGER** and **LOGICAL** data entities (that is, those for which no length or kind is specified).

Background Information

The specified size[1] applies to these data entities:

- **INTEGER** and **LOGICAL** specification statements with no length or kind specified.
- **FUNCTION** statements with no length or kind specified.
- Intrinsic functions that accept or return default **INTEGER** or **LOGICAL** arguments or return values unless you specify a length or kind in an **INTRINSIC** statement. Any specified length or kind must agree with the default size of the return value.
- Variables that are implicit integers or logicals.
- Integer and logical literal constants with no kind specified. If the value is too large to be represented by the number of bytes that you specified, the compiler chooses a size that is large enough. The range for 2-byte integers is -(2**15) to 2**15-1, for 4-byte integers is -(2**31) to 2**31-1, and for 8-byte integers is -(2**63) to 2**63-1.
- Typeless constants in integer or logical contexts.

Allowed sizes for *bytes* are:

- 2
- 4 (the default)
- 8

This option is intended to allow you to port programs unchanged from systems that have different default sizes for data. For example, you might need **-qintsize=2** for programs that are written for a 16-bit microprocessor or **-qintsize=8** for programs that are written for a CRAY computer. The default value of 4 for this option is suitable for code that is written specifically for many 32-bit computers. Note that specifying the **-q64** compiler option does not affect the default setting for **-qintsize**.

Restrictions

This option is not intended as a general-purpose method for increasing the sizes of data entities. Its use is limited to maintaining compatibility with code that is written for other systems.

You might need to add **PARAMETER** statements to give explicit lengths to constants that you pass as arguments.

1. In Fortran 90/95 terminology, these values are referred to as *kind type parameters*.

Examples

In the following example, note how variables, literal constants, intrinsics, arithmetic operators, and input/output operations all handle the changed default integer size.

```
@PROCESS INTSIZE(8)
      PROGRAM INTSIZETEST
      INTEGER I
      I = -9223372036854775807    ! I is big enough to hold this constant.
      J = ABS(I)                  ! So is implicit integer J.
      IF (I .NE. J) THEN
        PRINT *, I, '.NE.', J
      END IF
      END
```

The following example only works with the default size for integers:

```
      CALL SUB(17)
      END

      SUBROUTINE SUB(I)
      INTEGER(4) I                ! But INTSIZE may change "17"
                                  ! to INTEGER(2) or INTEGER(8).
      ...
      END
```

If you change the default value, you must either declare the variable **I** as **INTEGER** instead of **INTEGER(4)** or give a length to the actual argument, as follows:

```
@PROCESS INTSIZE(8)
      INTEGER(4) X
      PARAMETER(X=17)
      CALL SUB(X)      ! Use a parameter with the right length, or
      CALL SUB(17_4)   ! use a constant with the right kind.
      END
```

Related Information

See “-qrealsize Option” on page 222 and *Type Parameters and Specifiers* in the *XL Fortran Enterprise Edition for AIX Language Reference*.

-qipa Option

Syntax

`-qipa[=suboptions] | -qnoipa`

Enhances **-O** optimization by doing detailed analysis across procedures (interprocedural analysis or IPA).

You must also specify the **-O**, **-O2**, **-O3**, **-O4**, or **-O5** option when you specify **-qipa**. (Specifying the **-O5** option is equivalent to specifying the **-O4** option plus **-qipa=level=2**.) For additional performance benefits, you can also specify the **-Q** option. The **-qipa** option extends the area that is examined during optimization and inlining from a single procedure to multiple procedures (possibly in different source files) and the linkage between them.

You can fine-tune the optimizations that are performed by specifying suboptions.

To use this option, the necessary steps are:

1. Do preliminary performance analysis and tuning before compiling with the **-qipa** option. This is necessary because interprocedural analysis uses a two-phase mechanism, a compile-time and a link-time phase, which increases link time. (You can use the **noobject** suboption to reduce this overhead.)
2. Specify the **-qipa** option on both the compile and link steps of the entire application or on as much of it as possible. Specify suboptions to indicate what assumptions to make about the parts of the program that are not compiled with **-qipa**. (If your application contains C or C++ code compiled with IBM XL C/C++ compilers, you must compile with the **-qipa** option to allow for additional optimization opportunities at link time.)

   During compilation, the compiler stores interprocedural analysis information in the **.o** file. During linking, the **-qipa** option causes a complete reoptimization of the entire application.

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

Arguments

IPA uses the following suboptions during its compile-time phase:

object | **noobject**

Specifies whether to include standard object code in the object files. Specifying the **noobject** suboption can substantially reduce overall compilation time, by not generating object code during the first IPA phase. Note that if you specify **-S** with **noobject**, **noobject** will be ignored.

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

If your program contains object files created with the **noobject** suboption, you must use the **-qipa** option to compile any files containing an entry point (the main program for an executable program or an exported procedure for a library) before linking your program with **-qipa**.

IPA uses the following suboptions during its link-time phase:

exits=*procedure_names*

Specifies a list of procedures, each of which always ends the program. The compiler can optimize calls to these procedures (for example, by eliminating save/restore sequences), because the calls never return to the program. These procedures must not call any other parts of the program that are compiled with **-qipa**.

inline=*inline-options*
The **-qipa=inline=** command can take a colon-separated list of inline options, as listed below:

inline=auto | noauto
Specifies whether to automatically inline procedures.

inline=limit=*number*
Changes the size limits that the **-Q** option uses to determine how much inline expansion to do. This established "limit" is the size below which the calling procedure must remain. *number* is the optimizer's approximation of the number of bytes of code that will be generated. Larger values for this number allow the compiler to inline larger subprograms, more subprogram calls, or both. This argument is implemented only when **inline=auto** is on.

inline=*procedure_names*
Specifies a list of procedures to try to inline.

inline=threshold=*number*
Specifies the upper size limit on procedures to be inlined, where *number* is a value as defined under the inline suboption "limit". This argument is implemented only when "inline=auto" is on.

Note: By default, the compiler will try to inline all procedures, not just those that you specified with the **inline=***procedure_names* suboption. If you want to turn on inlining for only certain procedures, specify **inline=noauto** after you specify **inline=***procedure_names*. (You must specify the suboptions in this order.) For example, to turn off inlining for all procedures other than for **sub1**, specify **-qipa=inline=sub1:inline=noauto.**

isolated=*procedure_names*
Specifies a comma-separated list of procedures that are not compiled with **-qipa**. Procedures that you specify as "isolated" or procedures within their call chains cannot refer directly to any global variable.

level=*level*
Determines the amount of interprocedural analysis and optimization that is performed:

- **0** Does only minimal interprocedural analysis and optimization.
- **1** Turns on inlining, limited alias analysis, and limited call-site tailoring.
- **2** Full interprocedural data flow and alias analysis. Specifying **-O5** is equivalent to specifying **-O4** and **-qipa=level=2**.

The default level is **1**.

list=[*filename* **| short | long]**

Specifies an output listing file name during the link phase, in the event that an object listing has been requested using either the **-qlist** or the **-qipa=list** compiler option and allows the user to direct the type of output. If you do not specify the *filename* suboption, the default file name is "a.lst".

If you specify **short**, the Object File Map, Source File Map, and Global Symbols Map sections are included. If you specify **long**, the preceding sections appear in addition to the Object Resolution Warnings, Object Reference Map, Inliner Report, and Partition Map sections.

If you specify the **-qipa** and **-qlist** options together, IPA generates an `a.lst` file that overwrites any existing `a.lst` file. If you have a source file named `a.f`, the IPA listing will overwrite the regular compiler listing `a.lst`. You can use the **list=***filename* suboption to specify an alternative listing file name.

lowfreq=*procedure_names*
: Specifies a list of procedures that are likely to be called infrequently during the course of a typical program run. For example, procedures for initialization and cleanup might only be called once, and debugging procedures might not be called at all in a production-level program. The compiler can make other parts of the program faster by doing less optimization for calls to these procedures.

missing={unknown | safe | isolated | pure}
: Specifies the interprocedural behavior of procedures that are not compiled with **-qipa** and are not explicitly named in an **unknown**, **safe**, **isolated**, or **pure** suboption. The default is to assume **unknown**, which greatly restricts the amount of interprocedural optimization for calls to those procedures.

noinline=*procedure_names*
: Specifies a list of procedures that are not to be inlined.

partition={small | medium | large}
: Specifies the size of the regions within the program to analyze. Larger partitions contain more procedures, which result in better interprocedural analysis but require more storage to optimize. Reduce the partition size if compilation takes too long because of paging.

pdfname=[*filename***]**

: Specifies the name of the profile data file containing the **PDF** profiling information. If you do not specify a ***filename***, the default file name is **__pdf**. The profile is placed in the current working directory or in the directory that the **PDFDIR** environment variable names. This allows the programmer to do simultaneous runs of multiple executables using the same **PDFDIR**. This is especially useful when tuning with **PDF** on dynamic libraries. (See “-qpdf Option” on page 210 for more information on tuning optimizations.)

pure=*procedure_names*
: Specifies a list of procedures that are not compiled with **-qipa**. Any procedure that you specified as “pure” must be “isolated” and “safe”. It must not alter the internal state nor have side-effects, which are defined as potentially altering any data object visible to the caller.

safe=*procedure_names*
: Specifies a list of procedures that are not compiled with **-qipa**. Any procedure that you specified as “safe” may modify global variables and dummy arguments. No calls to procedures that are compiled with **-qipa** may be made from within a “safe” procedure’s call chain.

stdexits | nostdexits
: Specifies that certain predefined routines can be optimized as with the **exits** suboption. The procedures are: **abort**, **exit**, **_exit**, and **_assert**.

threads[=N] | **nothreads**
: **threads[=N]** runs the number of parallel threads that are available, or as specified by *N*. *N* must be a positive integer. **nothreads** does not run any parallel threads. This is equivalent to running one serial thread.

unknown=*procedure_names*
: Specifies a list of procedures that are not compiled with **-qipa**. Any procedure specified as "unknown" may make calls to other parts of the program compiled with **-qipa** and modify global variables and dummy arguments.

The primary use of **isolated**, **missing**, **pure**, **safe**, and **unknown** is to specify how much optimization can safely be performed on calls to library routines that are not compiled with **-qipa**.

The following compiler options have an effect on the link-time phase of **-qipa**:

-qlibansi | **-qnolibansi**
: Assumes that all functions with the name of an ANSI C defined library function are, in fact, the library functions.

-qlibessl | **-qnolibessl**
: Assumes that all functions with the name of an ESSL defined library function are, in fact, the library functions.

-qlibposix | **-qnolibposix**
: Assumes that all functions with the name of a POSIX 1003.1 defined library function are, in fact, the system functions.

-qthreaded
: Assumes that the compiler will attempt to generate thread-safe code.

Applicable Product Levels

This option is similar but not identical to the **-qipa** option of XL Fortran Version 3. If you have makefiles that already contain the **-qipa** option, modify them as needed to use the new suboption names.

Rules

Regular expressions are supported for the following suboptions:

exits
inline
lowfreq
noinline
pure
safe
unknown

Syntax rules for regular expressions are described below.

Table 15. Regular expression syntax

Expression	Description
`string`	Matches any of the characters specified in `string`. For example, `test` will match `testimony`, `latest`, `intestine`.
`^string`	Matches the pattern specified by `string` only if it occurs at the beginning of a line.

Table 15. Regular expression syntax (continued)

Expression	Description
`string$`	Matches the pattern specified by `string` only if it occurs at the end of a line.
`str.ing`	Matches any character. For example, `t.st` will match `test`, `tast`, `tZst`, and `t1st`.
`string\.$`	The backslash (\) can be used to escape special characters so that you can match for the character. For example, if you want to find those lines ending with a period, the expression `.$` would show all lines that had at least one character. Specify `\.$` to escape the period (.).
`[string]`	Matches any of the characters specified in `string`. For example, `t[a-g123]st` matches `tast` and `test`, but not `t-st` or `tAst`.
`[^string]`	Does not match any of the characters specified in `string`. For example, `t[^a-zA-Z]st` matches `t1st`, `t-st`, and `t,st` but not `test` or `tYst`.
`string*`	Matches zero or more occurrences of the pattern specified by `string`. For example, `te*st` will match `tst`, `test`, and `teeeeeest`.
`string+`	Matches one or more occurrences of the pattern specified by `string`. For example, `t(es)+t` matches `test`, `tesest`, but not `tt`.
`string?`	Matches zero or more occurrences of the pattern specified by `string`. For example, `te?st` matches either `tst` or `test`.
`string{m,n}`	Matches between `m` and `n` occurrence(s) of the pattern specified by `string`. For example, `a{2}` matches `aa`, `b{1,4}` matches `b`, `bb`, `bbb`, and `bbbb`.
`string1 | string2`	Matches the pattern specified by either `string1` or `string2`. For example, `s | o` matches both characters `s` and `o`.

Since only function names are being considered, the regular expressions are automatically bracketed with the ^ and $ characters. For example, **-qipa=noinline=^foo$** is equivalent to **-qipa=noinline=foo.** Therefore, **-qipa=noinline=bar** ensures that **bar** is never inlined but **bar1**, **teebar**, and **barrel** may be inlined.

Examples

The following example shows how you might compile a set of files with interprocedural analysis:

```
xlf95 -O -qipa f.f
xlf95 -c -O3 *.f -qipa=noobject
xlf95 -o product *.o -qipa -O
```

The following example shows how you might link these same files with interprocedural analysis, using regular expressions to improve performance. This example assumes that function **user_abort** exits the program, and that routines **user_trace1**, **user_trace2**, and **user_trace3** are rarely called.

```
xlf95 -o product *.o -qipa=exit=user_abort:lowfreq=user_trace[123] -O
```

Related Information

See the "-O Option" on page 114, "-p Option" on page 118, and "-Q Option" on page 119.

-qkeepparm Option

Syntax

`-qkeepparm | -qnokeepparm`

Background Information

A procedure usually stores its incoming parameters on the stack at the entry point. When you compile code with optimization, however, the optimizer may remove the stores into the stack if it sees opportunities to do so.

Specifying the **-qkeepparm** compiler option ensures that the parameters are stored on the stack even when optimizing. This may negatively impact execution performance. This option then provides access to the values of incoming parameters to tools, such as debuggers, simply by preserving those values on the stack.

-qlanglvl Option

Syntax

```
-qlanglvl={suboption}
LANGLVL({suboption})
```

Determines which language standard (or superset, or subset of a standard) to consult for nonconformance. It identifies nonconforming source code and also options that allow such nonconformances.

Rules

The compiler issues a message with severity code **L** if it detects syntax that is not allowed by the language level that you specified.

Arguments

77std Accepts the language that the ANSI FORTRAN 77 standard specifies and reports anything else as an error.

90std Accepts the language that the ISO Fortran 90 standard specifies and reports anything else as an error.

90pure The same as **90std** except that it also reports errors for any obsolescent Fortran 90 features used.

90ext Obsolete suboption that is equivalent to **extended**.

95std Accepts the language that the ISO Fortran 95 standard specifies and reports anything else as an error.

95pure The same as **95std** except that it also reports errors for any obsolescent Fortran 95 features used.

extended Accepts the full Fortran 95 language standard and all extensions, effectively turning off language-level checking.

Defaults

The default is **-qlanglvl=extended.** Prior to XL Fortran Version 6.1, the default was **-qlanglvl=90ext**. The **90ext** suboption accepts the full Fortran 90 language standard plus all extensions (now including the Fortran 95 standard) and is equivalent to **extended**. However, the **90ext** suboption is now obsolete, and to avoid problems in the future, you should start using the **extended** suboption as soon as possible.

Restrictions

The **-qflag** option can override this option.

Examples

The following example contains source code that conforms to a mixture of Fortran standards:

```
!----------------------------------------------------------
! in free source form
program tt
    integer :: a(100,100), b(100), i
    real :: x, y
    ...
    goto (10, 20, 30), i
10 continue
    pause 'waiting for input'

20 continue
    y= gamma(x)

30 continue
    b = maxloc(a, dim=1, mask=a .lt 0)

end program
!----------------------------------------------------------
```

The following chart shows examples of how some **-qlanglvl** suboptions affect this sample program:

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

Related Information

See "-qflag Option" on page 162, "-qhalt Option" on page 170, and "-qsaa Option" on page 227.

The **langlvl** run-time option, which is described in "Setting Run-Time Options" on page 51, helps to locate run-time extensions that cannot be checked for at compile time.

-qlargepage Option

Syntax

`–qlargepage | –qnolargepage`

Indicates to the compiler that a program, designed to execute in a large page memory environment, can take advantage of large 16 MB pages provided on POWER4 and higher based systems. When using **–qlargepage** with a program designed for a large page environment, an increase in performance can occur. See *AIX Performance Management Guide* for more information on using large page support.

Note: When using AIX 5.1, performance degradation can occur if there are too many programs attempting to access large pages at the same time. Performance degradation can also occur if you attempt to use **–qlargepage** without meeting the hardware requirements. Use this option with discretion.

The **–qlargepage** compiler option only takes effect with an optimization level that turns on the optimizer; a higher optimization level may do more.

-qlibansi Option

Related Information

See "-qipa Option" on page 182.

-qlibessl Option

Related Information

See "-qipa Option" on page 182.

-qlibposix Option

Related Information

See "-qipa Option" on page 182.

-qlist Option

Syntax

```
-qlist | -qnolist
LIST | NOLIST
```

Specifies whether to produce the object section of the listing.

You can use the object listing to help understand the performance characteristics of the generated code and to diagnose execution problems.

If you specify the **-qipa** and **-qlist** options together, IPA generates an `a.lst` file that overwrites any existing `a.lst` file. If you have a source file named `a.f`, the IPA listing will overwrite the regular compiler listing `a.lst`. To avoid this, use the **list=***filename* suboption of **-qipa** to generate an alternative listing.

Related Information

See "Options That Control Listings and Messages" on page 77, "Object Section" on page 393, and "-S Option" on page 269.

-qlistopt Option

Syntax

```
-qlistopt | -qnolistopt
LISTOPT | NOLISTOPT
```

Determines whether to show the setting of every compiler option in the listing file or only selected options. These selected options include those specified on the command line or directives plus some that are always put in the listing.

You can use the option listing during debugging to check whether a problem occurs under a particular combination of compiler options or during performance testing to record the optimization options in effect for a particular compilation.

Rules

Options that are always displayed in the listing are:

- All “on/off” options that are on by default: for example, **-qobject**
- All “on/off” options that are explicitly turned off through the configuration file, command-line options, or **@PROCESS** directives
- All options that take arbitrary numeric arguments (typically sizes)
- All options that have multiple suboptions

Related Information

See “Options That Control Listings and Messages” on page 77 and “Options Section” on page 389.

-qlm Option

Syntax

-qlm | -qnolm
LM | NOLM

Enables or disables the license management control system (LM). If you do not specify the **-qnolm** option, LM is enabled by default.

Use the **-qnolm** compiler option on the command line when compiling one program, or place the option in your configuration file (**xlf.cfg**) if you want LM disabled by default.

Related Information

See "Tracking Use of the Compiler" on page 38.

-qlog4 Option

Syntax

```
-qlog4 | -qnolog4
LOG4 | NOLOG4
```

Specifies whether the result of a logical operation with logical operands is a **LOGICAL(4)** or is a **LOGICAL** with the maximum length of the operands.

You can use this option to port code that was originally written for the IBM VS FORTRAN compiler.

Arguments

-qlog4 makes the result always a **LOGICAL(4)**, while **-qnolog4** makes it depend on the lengths of the operands.

Restrictions

If you use **-qintsize** to change the default size of logicals, **-qlog4** is ignored.

-qmaxmem Option

Syntax

```
-qmaxmem=Kbytes
MAXMEM(Kbytes)
```

Limits the amount of memory that the compiler allocates while performing specific, memory-intensive optimizations to the specified number of kilobytes. A value of -1 allows optimization to take as much memory as it needs without checking for limits.

Defaults

At the **-O2** optimization level, the default **-qmaxmem** setting is 2048 KB. At the **-O3** optimization level, the default setting is unlimited (-1).

Rules

If the specified amount of memory is insufficient for the compiler to compute a particular optimization, the compiler issues a message and reduces the degree of optimization.

This option has no effect except in combination with the **-O** option.

When compiling with **-O2**, you only need to increase the limit if a compile-time message instructs you to do so. When compiling with **-O3**, you might need to establish a limit if compilation stops because the machine runs out of storage; start with a value of 2048 or higher, and decrease it if the compilation continues to require too much storage.

Notes:

1. Reduced optimization does not necessarily mean that the resulting program will be slower. It only means that the compiler cannot finish looking for opportunities to improve performance.
2. Increasing the limit does not necessarily mean that the resulting program will be faster. It only means that the compiler is better able to find opportunities to improve performance if they exist.
3. Setting a large limit has no negative effect when compiling source files for which the compiler does not need to use so much memory during optimization.
4. As an alternative to raising the memory limit, you can sometimes move the most complicated calculations into procedures that are then small enough to be fully analyzed.
5. Not all memory-intensive compilation stages can be limited.
6. Only the optimizations done for **-O2** and **-O3** can be limited; **-O4** and **-O5** optimizations cannot be limited.
7. The **-O4** and **-O5** optimizations may also use a file in the /tmp directory. This is not limited by the **-qmaxmem** setting.
8. Some optimizations back off automatically if they would exceed the maximum available address space, but not if they would exceed the paging space available at that time, which depends on machine workload.

Restrictions

Depending on the source file being compiled, the size of subprograms in the source code, the machine configuration, and the workload on the system, setting the limit too high might fill up the paging space. In particular, a value of -1 can fill up the storage of even a well-equipped machine.

Related Information

See "-O Option" on page 114 and "Optimizing XL Fortran Programs" on page 305.

-qmbcs Option

Syntax

```
-qmbcs | -qnombcs
MBCS | NOMBCS
```

Indicates to the compiler whether character literal constants, Hollerith constants, H edit descriptors, and character string edit descriptors can contain Multibyte Character Set (MBCS) or Unicode characters.

This option is intended for applications that must deal with data in a multibyte language, such as Japanese.

To process the multibyte data correctly at run time, set the locale (through the **LANG** environment variable or a call to the **libc setlocale** routine) to the same value as during compilation.

Rules

Each byte of a multibyte character is counted as a column.

Restrictions

To read or write Unicode data, set the locale value to **UNIVERSAL** at run time. If you do not set the locale, you might not be able to interchange data with Unicode-enabled applications.

-qmixed Option

Syntax

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

This is the long form of the “-U Option” on page 271.

-qmoddir Option

Syntax

`-qmoddir=`*`directory`*

Specifies the location for any module (**.mod**) files that the compiler writes.

Defaults

If you do not specify **-qmoddir**, the **.mod** files are placed in the current directory.

To read the **.mod** files from this directory when compiling files that reference the modules, use the “`-I` Option” on page 109.

Related Information

See “XL Fortran Output Files” on page 34.

Modules are a Fortran 90/95 feature and are explained in the Modules section of the *XL Fortran Enterprise Edition for AIX Language Reference*.

-qmodule Option

Syntax

```
-qmodule=mangle81
```

Specifies that the compiler should use the XL Fortran Version 8.1 naming convention for non-intrinsic module files.

This option allows you to produce modules and their associated object files with the Version 9.1 compiler and link these object files with others compiled with the Version 8.1 compiler, or earlier.

Related Information

Modules are a Fortran 90/95 feature and are explained in the Modules section of the *XL Fortran Enterprise Edition for AIX Language Reference*.

See also "Conventions for XL Fortran External Names" on page 345 and "Avoiding Naming Conflicts during Linking" on page 47.

-qnoprint Option

Syntax

```
-qnoprint
```

Prevents the compiler from creating the listing file, regardless of the settings of other listing options.

Specifying **-qnoprint** on the command line enables you to put other listing options in a configuration file or on **@PROCESS** directives and still prevent the listing file from being created.

Rules

A listing file is usually created when you specify any of the following options: **-qattr**, **-qlist**, **-qlistopt**, **-qphsinfo**, **-qsource**, **-qreport**, or **-qxref**. **-qnoprint** prevents the listing file from being created by changing its name to **/dev/null**, a device that discards any data that is written to it.

Related Information

See "Options That Control Listings and Messages" on page 77.

-qnullterm Option

Syntax

-qnullterm | **-qnonullterm**
NULLTERM | **NONULLTERM**

Appends a null character to each character constant expression that is passed as a dummy argument, to make it more convenient to pass strings to C functions.

This option allows you to pass strings to C functions without having to add a null character to each string argument.

Background Information

This option affects arguments that are composed of any of the following objects: basic character constants; concatenations of multiple character constants; named constants of type character; Hollerith constants; binary, octal, or hexadecimal typeless constants when an interface block is available; or any character expression composed entirely of these objects. The result values from the **CHAR** and **ACHAR** intrinsic functions also have a null character added to them if the arguments to the intrinsic function are initialization expressions.

Rules

This option does not change the length of the dummy argument, which is defined by the additional length argument that is passed as part of the XL Fortran calling convention.

Restrictions

This option affects those arguments that are passed with or without the **%REF** built-in function, but it does not affect those that are passed by value. This option does not affect character expressions in input and output statements.

Examples

Here are two calls to the same C function, one with and one without the option:

```
      @PROCESS NONULLTERM
      SUBROUTINE CALL_C_1
        CHARACTER*9, PARAMETER :: HOME = "/home/luc"
! Call the libc routine mkdir() to create some directories.
        CALL mkdir ("/home/luc/testfiles\0", %val(448))
! Call the libc routine unlink() to remove a file in the home directory.
        CALL unlink (HOME // "/.hushlogin" // CHAR(0))
      END SUBROUTINE

      @PROCESS NULLTERM
      SUBROUTINE CALL_C_2
        CHARACTER*9, PARAMETER :: HOME = "/home/luc"
! With the option, there is no need to worry about the trailing null
! for each string argument.
        CALL mkdir ("/home/luc/testfiles", %val(448))
        CALL unlink (HOME // "/.hushlogin")
      END SUBROUTINE
!
```

Related Information

See “Passing Character Types between Languages” on page 351.

-qobject Option

Syntax

```
-qOBJect | -qNOOBJect
OBJect | NOOBJect
```

Specifies whether to produce an object file or to stop immediately after checking the syntax of the source files.

When debugging a large program that takes a long time to compile, you might want to use the **-qnoobject** option. It allows you to quickly check the syntax of a program without incurring the overhead of code generation. The **.lst** file is still produced, so you can get diagnostic information to begin debugging.

After fixing any program errors, you can change back to the default (**-qobject**) to test whether the program works correctly. If it does not work correctly, compile with the **-g** option for interactive debugging.

Restrictions

The **-qhalt** option can override the **-qobject** option, and **-qnoobject** can override **-qhalt**.

Related Information

See "Options That Control Listings and Messages" on page 77 and "Object Section" on page 393.

"The Compiler Phases" on page 411 gives some technical information about the compiler phases.

-qonetrip Option

Syntax

```
-qonetrip | -qnoonetrip
ONETRIP | NOONETRIP
```

This is the long form of the “-1 Option” on page 92.

-qoptimize Option

Syntax

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

This is the long form of the “-O Option” on page 114.

-qpdf Option

Syntax

```
-qpdf{1|2}
```

Tunes optimizations through *profile-directed feedback* (PDF), where results from sample program execution are used to improve optimization near conditional branches and in frequently executed code sections.

To use PDF, follow these steps:

1. Compile some or all of the source files in a program with the **-qpdf1** option. You need to specify the **-O2** option, or preferably the **-O3**, **-O4**, or **-O5** option, for optimization. Pay special attention to the compiler options that you use to compile the files, because you will need to use the same options later.

   In a large application, concentrate on those areas of the code that can benefit most from optimization. You do not need to compile all of the application's code with the **-qpdf1** option.
2. Run the program all the way through using a typical data set. The program records profiling information when it finishes. You can run the program multiple times with different data sets, and the profiling information is accumulated to provide an accurate count of how often branches are taken and blocks of code are executed.

   Important: Use data that is representative of the data that will be used during a normal run of your finished program.
3. Relink your program using the same compiler options as before, but change **-qpdf1** to **-qpdf2**. Remember that **-L**, **-l**, and some others are linker options, and you can change them at this point. In this second compilation, the accumulated profiling information is used to fine-tune the optimizations. The resulting program contains no profiling overhead and runs at full speed.

For best performance, use the **-O3**, **-O4**, or **-O5** option with all compilations when you use PDF (as in the example above). If your application contains C or C++ code compiled with IBM XL C/C+ compilers, you can achieve additional PDF optimization by specifying the **-qpdf1** and **-qpdf2** options available on those compilers. Combining **-qpdf1**/**-qpdf2** and **-qipa** or **-O5** options (that is, link with IPA) on all Fortran and C/C++ code will lead to maximum PDF information being available for optimization.

Rules

The profile is placed in the current working directory or in the directory that the **PDFDIR** environment variable names, if that variable is set.

To avoid wasting compilation and execution time, make sure that the **PDFDIR** environment variable is set to an absolute path. Otherwise, you might run the application from the wrong directory, and it will not be able to locate the profile data files. When that happens, the program may not be optimized correctly or may be stopped by a segmentation fault. A segmentation fault might also happen if you change the value of the **PDFDIR** variable and execute the application before finishing the PDF process.

Background Information

Because this option requires compiling the entire application twice, it is intended to be used after other debugging and tuning is finished, as one of the last steps before putting the application into production.

Restrictions

- PDF optimizations also require at least the **-O2** optimization level.
- You must compile the main program with PDF for profiling information to be collected at run time.
- Do not compile or run two different applications that use the same **PDFDIR** directory at the same time, unless you have used the **-qipa=pdfname** suboption to distinguish the sets of profiling information.
- You must use the same set of compiler options at all compilation steps for a particular program. Otherwise, PDF cannot optimize your program correctly and may even slow it down. All compiler settings must be the same, including any supplied by configuration files.
- The **-qpdf** option affects only programs whose source files have been compiled using XL Fortran Version 7.1.1 or later. Avoid mixing object files that were compiled with the **-qpdf** option on previous compilers, with those compiled with the **-qpdf** option on XL Fortran compilers at version 7.1.1 or later.
- If **-qipa** is not invoked either directly or through other options, **-qpdf1** and **-qpdf2** will invoke the **-qipa=level=0** option.
- If you do compile a program with **-qpdf1**, remember that it will generate profiling information when it runs, which involves some performance overhead. This overhead goes away when you recompile with **-qpdf2** or with no PDF at all.

The following commands, in the directory **/usr/lpp/xlf/bin**, are available for managing the **PDFDIR** directory:

`cleanpdf [`*pathname*`]`	Removes all profiling information from the *pathname* directory; or if *pathname* is not specified, from the **PDFDIR** directory; or if **PDFDIR** is not set, from the current directory. Removing the profiling information reduces the run-time overhead if you change the program and then go through the PDF process again. Run this program after compiling with **-qpdf2** or after finishing with the PDF process for a particular application. If you continue using PDF with an application after running **cleanpdf**, you must recompile all the files with **-qpdf1**.

`mergepdf`

Generates a single pdf record from 2 or more input pdf records. All pdf records must come from the same executable.

mergepdf automatically scales each pdf record (that is, file) based on its "weight". A scaling ratio can be specified for each pdf record, so that more important training runs can be weighted heavier than less important ones. The syntax for **mergepdf** is:

```
mergepdf [-r1] record1 [-r2] record2 ... -outputrecname [-n] [-v]
```

where:

-r
The scaling ratio for the pdf record. If -r is not specified for an input record, the default ratio is 1.0 and no external scaling is applied. The scaling ratio must be greater than or equal to zero, and can be a floating point number or an integer.

record
The input file, or the directory that contains the pdf profile.

-o output_recordname
The pdf output directory name, or a file name that **mergepdf** will write the merged pdf record to. If a directory is specified, it must exist before you run the command.

-n
Do not normalize the pdf records. By default, records are normalized based on an internally calculated ratio for each profile before applying any user-specified ratio with **-r**. When **-n** is specified, the pdf records are scaled by the user-specified ratio **-r**. If **-r** is not specified, the pdf records are not scaled at all.

-v
Verbose mode displays the internal and user-specified scaling ratio used.

`resetpdf [`*pathname*`]`

Same as **cleanpdf** [*pathname*], described above. This command is provided for compatibility with the previous version.

`showpdf`

Displays the call and block counts for all procedures executed in a program run. To use this command, you must first compile your application specifying both **-qpdf1** and **-qshowpdf** compiler options

Examples

Here is a simple example:

```
# Set the PDFDIR variable.
  export PDFDIR=$HOME/project_dir
# Compile all files with -qpdf1.
  xlf95 -qpdf1 -O3 file1.f file2.f file3.f
# Run with one set of input data.
  a.out <sample.data
# Recompile all files with -qpdf2.
  xlf95 -qpdf2 -O3 file1.f file2.f file3.f
# The program should now run faster than without PDF if
# the sample data is typical.
```

Here is a more elaborate example:

```
# Set the PDFDIR variable.
  export PDFDIR=$HOME/project_dir
# Compile most of the files with -qpdf1.
  xlf95 -qpdf1 -O3 -c file1.f file2.f file3.f
# This file is not so important to optimize.
  xlf95 -c file4.f
# Non-PDF object files such as file4.o can be linked in.
  xlf95 -qpdf1 file1.o file2.o file3.o file4.o
```

```
# Run several times with different input data.
  a.out <polar_orbit.data
  a.out <elliptical_orbit.data
  a.out <geosynchronous_orbit.data
# Do not need to recompile the source of non-PDF object files (file4.f).
  xlf95 -qpdf2 -O3 file1.f file2.f file3.f
# Link all the object files into the final application.
  xlf95 file1.o file2.o file3.o file4.o
```

Related Information

See “XL Fortran Input Files” on page 33, “XL Fortran Output Files” on page 34, “Using Profile-directed Feedback (PDF)” on page 315, and “Optimizing Conditional Branching” on page 316.

-qphsinfo Option

Syntax

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

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

The output takes the form *number1/number2* for each phase where *number1* represents the CPU time used by the compiler and *number2* represents the total of the compiler time and the time that the CPU spends handling system calls.

Examples

To compile **app.f**, which consists of 3 compilation units, and report the time taken for each phase of the compilation, enter:

```
xlf90 app.f  -qphsinfo
```

The output will look similar to:

```
FORTRAN phase 1 ftphas1      TIME =  0.000 / 0.000
** m_module   === End of Compilation 1 ===
FORTRAN phase 1 ftphas1      TIME =  0.000 / 0.000
** testassign   === End of Compilation 2 ===
FORTRAN phase 1 ftphas1      TIME =  0.000 / 0.010
** dataassign   === End of Compilation 3 ===
HOT       - Phase Ends;   0.000/  0.000
HOT       - Phase Ends;   0.000/  0.000
HOT       - Phase Ends;   0.000/  0.000
W-TRANS   - Phase Ends;   0.000/  0.010
OPTIMIZ   - Phase Ends;   0.000/  0.000
REGALLO   - Phase Ends;   0.000/  0.000
AS        - Phase Ends;   0.000/  0.000
W-TRANS   - Phase Ends;   0.000/  0.000
OPTIMIZ   - Phase Ends;   0.000/  0.000
REGALLO   - Phase Ends;   0.000/  0.000
AS        - Phase Ends;   0.000/  0.000
W-TRANS   - Phase Ends;   0.000/  0.000
OPTIMIZ   - Phase Ends;   0.000/  0.000
REGALLO   - Phase Ends;   0.000/  0.000
AS        - Phase Ends;   0.000/  0.000
1501-510  Compilation successful for file app.f.
```

Each phase is invoked three times, corresponding to each compilation unit. `FORTRAN` represents front-end parsing and semantic analysis, `HOT` loop transformations, `W-TRANS` intermediate language translation, `OPTIMIZ` high–level optimization, `REGALLO` register allocation and low–level optimization, and `AS` final assembly.

Compile **app.f** at the **-O4** optimization level with **-qphsinfo** specified:

```
xlf90 myprogram.f  -qphsinfo -O4
```

The following output results:

```
FORTRAN phase 1 ftphas1      TIME =  0.010 / 0.020
** m_module   === End of Compilation 1 ===
FORTRAN phase 1 ftphas1      TIME =  0.000 / 0.000
** testassign   === End of Compilation 2 ===
FORTRAN phase 1 ftphas1      TIME =  0.000 / 0.000
** dataassign   === End of Compilation 3 ===
HOT       - Phase Ends;   0.000/  0.000
HOT       - Phase Ends;   0.000/  0.000
HOT       - Phase Ends;   0.000/  0.000
IPA       - Phase Ends;   0.080/  0.100
1501-510  Compilation successful for file app.f.
IPA       - Phase Ends;   0.050/  0.070
W-TRANS   - Phase Ends;   0.010/  0.030
OPTIMIZ   - Phase Ends;   0.020/  0.020
REGALLO   - Phase Ends;   0.040/  0.040
AS        - Phase Ends;   0.000/  0.000
```

Note that during the IPA (interprocedural analysis) optimization phases, the program has resulted in one compilation unit; that is, all procedures have been inlined.

Related Information

"The Compiler Phases" on page 411.

-qpic Option

Syntax

```
-qpic[=suboptions]
```

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

Arguments

small | large
The **small** suboption tells the compiler to assume that the size of the Table Of Contents be at most, 64K. The **large** suboption allows the size of the Table Of Contents to be larger than 64K. This suboption allows more addresses to be stored in the Table Of Contents. However, it does generate code that is usually larger than that generated by the **small** suboption.

-qpic=small is the default.

Specifying **-qpic=large** has the same effect as passing **-bbigtoc** to the **ld** command.

-qport Option

Syntax

```
-qport[=suboptions]| -qnoport
PORT[(suboptions)]| NOPORT
```

The **-qport** compiler option increases flexibility when porting programs to XL Fortran, providing a number of options to accommodate other Fortran language extensions. A particular suboption will always function independently of other **-qport** and compiler options.

Arguments

hexint | **nohexint**
: If you specify this option, typeless constant hexadecimal strings are converted to integers when passed as an actual argument to the **INT** intrinsic function. Typeless constant hexadecimal strings not passed as actual arguments to **INT** remain unaffected.

mod | **nomod**
: Specifying this option relaxes existing constraints on the **MOD** intrinsic function, allowing two arguments of the same data type parameter to be of different kind type parameters. The result will be of the same type as the argument, but with the larger kind type parameter value.

nullarg | **nonullarg**
: For an external or internal procedure reference, specifying this option causes the compiler to treat an empty argument, which is delimited by a left parenthesis and a comma, two commas, or a comma and a right parenthesis, as a null argument. This suboption has no effect if the argument list is empty.

Examples of empty arguments are:

```
call foo(,,z)

call foo(x,,z)

call foo(x,y,)
```

The following program includes a null argument.

Fortran program:

```
program nularg
real(4) res/0.0/
integer(4) rc
integer(4), external :: add
rc = add(%val(2), res, 3.14, 2.18,) ! The last argument is a
                                    ! null argument.
if (rc == 0) then
print *, "res = ", res
else
print *, "number of arguments is invalid."
endif
end program
```

C program:

```
int add(int a, float *res, float *b, float *c, float *d)
{
   int ret = 0;
   if (a == 2)
     *res = *b + *c;
```

```
    else if (a == 3)
      *res = (*b + *c + *d);
    else
      ret = 1;
    return (ret);
}
```

sce | **nosce**
By default, the compiler performs short circuit evaluation in selected logical expressions using XL Fortran rules. Specifying **sce** allows the compiler to use non-XL Fortran rules. The compiler will perform short circuit evaluation if the current rules allow it.

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

typlssarg | **notyplssarg**
Converts all typeless constants to default integers if the constants are actual arguments to an intrinsic procedure whose associated dummy arguments are of integer type. Dummy arguments associated with typeless actual arguments of noninteger type remain unaffected by this option.

Using this option may cause some intrinsic procedures to become mismatched in kinds. Specify **-qxlf77=intarg** to convert the kind to that of the longest argument.

Related Information

See the section on the **INT** and **MOD** intrinsic functions in the *XL Fortran Enterprise Edition for AIX Language Reference* for further information.

-qposition Option

Syntax

```
-qposition={appendold | appendunknown} ...
POSITION({APPENDOLD | APPENDUNKNOWN} ...)
```

Positions the file pointer at the end of the file when data is written after an **OPEN** statement with no **POSITION=** specifier and the corresponding **STATUS=** value (**OLD** or **UNKNOWN**) is specified.

Rules

The position becomes **APPEND** when the first I/O operation moves the file pointer if that I/O operation is a **WRITE** or **PRINT** statement. If it is a **BACKSPACE**, **ENDFILE**, **READ**, or **REWIND** statement instead, the position is **REWIND**.

Applicable Product Levels

The **appendunknown** suboption is the same as the XL Fortran Version 2 **append** suboption, but we recommend using **appendunknown** to avoid ambiguity.

-qposition=appendold:appendunknown provides compatibility with XL Fortran Version 1 and early Version 2 behavior. **-qposition=appendold** provides compatibility with XL Fortran Version 2.3 behavior.

Examples

In the following example, **OPEN** statements that do not specify a **POSITION=** specifier, but specify **STATUS='old'** will open the file as if **POSITION='append'** was specified.

```
xlf95 -qposition=appendold opens_old_files.f
```

In the following example, **OPEN** statements that do not specify a **POSITION=** specifier, but specify **STATUS='unknown'** will open the file as if **POSITION='append'** was specified.

```
xlf95 -qposition=appendunknown opens_unknown_files.f
```

In the following example, **OPEN** statements that do not specify a **POSITION=** specifier, but specify either **STATUS='old'** or **STATUS='unknown'** will open the file as if **POSITION='append'** was specified.

```
xlf95 -qposition=appendold:appendunknown opens_many_files.f
```

Related Information

See "File Positioning" on page 327 and the section on the **OPEN** statement in the *XL Fortran Enterprise Edition for AIX Language Reference*.

-qprefetch Option

Syntax

-qprefetch | -qnoprefetch

Instructs the compiler to insert the prefetch instructions automatically where there are opportunities to improve code performance.

Related Information

For more information on prefetch directives, see **PREFETCH directives** in the *XL Fortran Enterprise Edition for AIX Language Reference* and *The POWER4 Processor Introduction and Tuning Guide*. To selectively control prefetch directives using trigger constants, see the “-qdirective Option” on page 148.

-qqcount Option

Syntax

```
-qqcount | -qnoqcount
QCOUNT | NOQCOUNT
```

Accepts the **Q** character-count edit descriptor (**Q**) as well as the extended-precision **Q** edit descriptor (**Q***w.d*). With **-qnoqcount**, all **Q** edit descriptors are interpreted as the extended-precision **Q** edit descriptor.

Rules

The compiler interprets a **Q** edit descriptor as one or the other depending on its syntax and issues a warning if it cannot determine which one is specified.

Related Information

See *Q (Character Count) Editing* in the *XL Fortran Enterprise Edition for AIX Language Reference*.

-qrealsize Option

Syntax

```
-qrealsize=bytes
REALSIZE(bytes)
```

Sets the default size of **REAL**, **DOUBLE PRECISION**, **COMPLEX**, and **DOUBLE COMPLEX** values.

This option is intended for maintaining compatibility with code that is written for other systems. You may find it useful as an alternative to **-qautodbl** in some situations.

Rules

The option affects the sizes[2] of constants, variables, derived type components, and functions (which include intrinsic functions) for which no kind type parameter is specified. Objects that are declared with a kind type parameter or length, such as **REAL(4)** or **COMPLEX*16**, are not affected.

Arguments

The allowed values for *bytes* are:

- 4 (the default)
- 8

Results

This option determines the sizes of affected objects as follows:

```
Data Object       REALSIZE(4) in Effect       REALSIZE(8) in Effect
-------------------------------------------------------------------
1.2               REAL(4)                      REAL(8)
1.2e0             REAL(4)                      REAL(8)
1.2d0             REAL(8)                      REAL(16)
1.2q0             REAL(16)                     REAL(16)

REAL              REAL(4)                      REAL(8)
DOUBLE PRECISION  REAL(8)                      REAL(16)
COMPLEX           COMPLEX(4)                   COMPLEX(8)
DOUBLE COMPLEX    COMPLEX(8)                   COMPLEX(16)
```

Similar rules apply to intrinsic functions:

- If an intrinsic function has no type declaration, its argument and return types may be changed by the **-qrealsize** setting.
- Any type declaration for an intrinsic function must agree with the default size of the return value.

This option is intended to allow you to port programs unchanged from systems that have different default sizes for data. For example, you might need **-qrealsize=8** for programs that are written for a CRAY computer. The default value of 4 for this option is suitable for programs that are written specifically for many 32-bit computers.

Setting **-qrealsize** to 8 overrides the setting of the **-qdpc** option.

2. In Fortran 90/95 terminology, these values are referred to as *kind type parameters*.

Examples

This example shows how changing the **-qrealsize** setting transforms some typical entities:

```
@PROCESS REALSIZE(8)
         REAL R                  ! treated as a real(8)
         REAL(8) R8              ! treated as a real(8)
         DOUBLE PRECISION DP     ! treated as a real(16)
         DOUBLE COMPLEX DC       ! treated as a complex(16)
         COMPLEX(4) C            ! treated as a complex(4)
         PRINT *,DSIN(DP)        ! treated as qsin(real(16))
! Note: we cannot get dsin(r8) because dsin is being treated as qsin.
         END
```

Specifying **-qrealsize=8** affects intrinsic functions, such as **DABS**, as follows:

```
INTRINSIC DABS        ! Argument and return type become REAL(16).
DOUBLE PRECISION DABS ! OK, because DOUBLE PRECISION = REAL(16)
                      ! with -qrealsize=8 in effect.
REAL(16) DABS         ! OK, the declaration agrees with the option setting.
REAL(8) DABS          ! The declaration does not agree with the option
                      ! setting and is ignored.
```

Related Information

"-qintsize Option" on page 180 is a similar option that affects integer and logical objects. "-qautodbl Option" on page 134 is related to **-qrealsize**, although you cannot combine the options. When the **-qautodbl** option turns on automatic doubling, padding, or both, the **-qrealsize** option has no effect.

Type Parameters and Specifiers in the *XL Fortran Enterprise Edition for AIX Language Reference* discusses kind type parameters.

-qrecur Option

Syntax

```
-qrecur | -qnorecur
RECUR | NORECUR
```

Not recommended. Specifies whether external subprograms may be called recursively. For new programs, use the **RECURSIVE** keyword, which provides a standard-conforming way of using recursive procedures. If you specify the **-qrecur** option, the compiler must assume that any procedure could be recursive. Code generation for recursive procedures may be less efficient. Using the **RECURSIVE** keyword allows you to specify exactly which procedures are recursive.

Examples

```
! The following RECUR recursive function:

        @process recur
        function factorial (n)
        integer factorial
        if (n .eq. 0) then
           factorial = 1
        else
           factorial = n * factorial (n-1)
        end if
        end function factorial

! can be rewritten to use F90/F95 RECURSIVE/RESULT features:

        recursive function factorial (n) result (res)
        integer res
        if (n .eq. 0) then
           res = 1
        else
           res = n * factorial (n-1)
        end if
        end function factorial
```

Restrictions

If you use the **xlf**, **xlf_r**, **xlf_r7**, **f77**, or **fort77** command to compile programs that contain recursive calls, specify **-qnosave** to make the default storage class automatic.

-qreport Option

Syntax

```
-qreport[={smplist | hotlist}...]
-qnoreport
REPORT[({SMPLIST | HOTLIST}...)] NOREPORT
```

Determines whether to produce transformation reports showing how the program is parallelized and how loops are optimized.

You can use the **smplist** suboption to debug or tune the performance of SMP programs by examining the low-level transformations. You can see how the program deals with data and the automatic parallelization of loops. Comments within the listing tell you how the transformed program corresponds to the original source code and include information as to why certain loops were not parallelized.

You can use the **hotlist** suboption to generate a report showing how loops are transformed.

Arguments

smplist
: Produces a pseudo-Fortran listing that shows how the program is parallelized. This listing is produced before loop and other optimizations are performed. It includes messages that point out places in the program that can be modified to be more efficient. This report will only be produced if the **-qsmp** option is in effect.

hotlist
: Produces a pseudo-Fortran listing that shows how loops are transformed, to assist you in tuning the performance of all loops. This suboption is the default if you specify **-qreport** with no suboptions.

  In addition, if you specify the **-qreport=hotlist** option when the **-qsmp** option is in effect, a pseudo-Fortran listing will be produced that shows the calls to the SMP runtime and the procedures created for parallel constructs.

Background Information

The transformation listing is part of the compiler listing file.

Restrictions

Loop transformation and auto parallelization are done on the link step at a **-O5** (or **-qipa=level=2**) optimization level. The **-qreport** option will generate the report in the listing file on the link step.

You must specify the **-qsmp** or the **-qhot** option to generate a loop transformation listing. You must specify the **-qsmp** option to generate a parallel transformation listing or parallel performance messages.

The code that the listing shows is not intended to be compilable. Do not include any of this code in your own programs or explicitly call any of the internal routines whose names appear in the listing.

Examples

To produce a listing file that you can use to tune parallelization:

```
xlf_r -qsmp -O3 -qhot -qreport=smplist needs_tuning.f
```

To produce a listing file that you can use to tune both parallelization and loop performance:

```
xlf_r -qsmp -O3 -qhot -qreport=smplist:hotlist needs_tuning.f
```

To produce a listing file that you can use to tune only the performance of loops:

```
xlf95_r -O3 -qhot -qreport=hotlist needs_tuning.f
```

Related Information

See "-qpdf Option" on page 210.

-qsaa Option

Syntax

```
-qsaa | -qnosaa
SAA | NOSAA
```

Checks for conformance to the SAA FORTRAN language definition. It identifies nonconforming source code and also options that allow such nonconformances.

Rules

These warnings have a prefix of **(L)**, indicating a problem with the language level.

Restrictions

The **-qflag** option can override this option.

Related Information

Use the "-qlanglvl Option" on page 189 to check your code for conformance to international standards.

-qsave Option

Syntax

```
-qsave[={all|defaultinit}] | -qnosave
SAVE[({all|defaultinit})] NOSAVE
```

This specifies the default storage class for local variables.

If **-qsave=all** is specified, the default storage class is **STATIC**; if **-qnosave** is specified, the default storage class is **AUTOMATIC**; if **-qsave=defaultinit** is specified, the default storage class is **STATIC** for variables of derived type that have default initialization specified, and **AUTOMATIC** otherwise. The default suboption for the **-qsave** option is **all**. The two suboptions are mutually exclusive.

The default for this option depends on the invocation used. For example, you may need to specify **-qsave** to duplicate the behavior of FORTRAN 77 programs. The **xlf**, **xlf_r**, **xlf_r7**, **f77**, and **fort77** commands have **-qsave** listed as a default option in **/etc/xlf.cfg** to preserve the previous behavior.

The following example illustrates the impact of the **-qsave** option on derived data type:

```
PROGRAM P
  CALL SUB
  CALL SUB
END PROGRAM P

SUBROUTINE SUB
  LOGICAL, SAVE :: FIRST_TIME = .TRUE.
  STRUCTURE /S/
    INTEGER I/17/
  END STRUCTURE
  RECORD /S/ LOCAL_STRUCT
  INTEGER LOCAL_VAR

  IF (FIRST_TIME) THEN
    LOCAL_STRUCT.I = 13
    LOCAL_VAR = 19
    FIRST_TIME = .FALSE.
  ELSE
    ! Prints " 13" if compiled with -qsave or -qsave=all
    ! Prints " 13" if compiled with -qsave=defaultinit
    ! Prints " 17" if compiled with -qnosave
    PRINT *, LOCAL_STRUCT
    ! Prints " 19" if compiled with -qsave or -qsave=all
    ! Value of LOCAL_VAR is undefined otherwise
    PRINT *, LOCAL_VAR
  END IF
END SUBROUTINE SUB
```

Related Information

The **-qnosave** option is usually necessary for multi-threaded applications and subprograms that are compiled with the “-qrecur Option” on page 224.

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

-qsaveopt Option

Syntax

`-qsaveopt | -qnosaveopt`

Saves the command-line options used for compiling a source file, and other information, in the corresponding object file. The compilation must produce an object file for this option to take effect. Only one copy of the command-line options is saved, even though each object may contain multiple compilation units.

To list the options used, issue the **what -s** command on the object file. The following is listed:

```
opt source_type invocation_used compilation_options
```

For example, if the object file is **t.o**, the **what -s t.o** command may produce information similar to the following:

```
opt f /usr/bin/xlf/xlf90 -qlist -qsaveopt t.f
```

where **f** identifies the source used as Fortran, **/usr/bin/xlf/xlf90** shows the invocation command used, and **-qlist -qsaveopt** shows the compilation options.

-qsclk Option

Syntax

```
-qsclk[=centi | micro]
```

Specifies the resolution that the **SYSTEM_CLOCK** intrinsic procedure uses in a program. The default is centisecond resolution (**–qsclk=centi**). To use microsecond resolution, specify **–qsclk=micro**.

Related Information

See **SYSTEM_CLOCK** in the *XL Fortran Enterprise Edition for AIX Language Reference* for more information on returning integer data from a real-time clock.

-qshowpdf Option

Syntax

`-qshowpdf` | **`-qnoshowpdf`**

Used together with **-qpdf1** to add additional call and block count profiling information to an executable.

When specified together with **-qpdf1**, the compiler inserts additional profiling information into the compiled application to collect call and block counts for all procedures in the application. Running the compiled application will record the call and block counts to the **._pdf** file.

After you run your application with training data, you can retrieve the contents of the **._pdf** file with the **showpdf** utility. This utility is described in "-qpdf Option" on page 210.

-qsigtrap Option

Syntax

`-qsigtrap[=`*`trap_handler`*`]`

When you are compiling a file that contains a main program, this option sets up the specified trap handler to catch **SIGTRAP** exceptions. This option enables you to install a handler for **SIGTRAP** signals without calling the **SIGNAL** subprogram in the program.

Arguments

To enable the `xl__trce` trap handler, specify **-qsigtrap** without a handler name. To use a different trap handler, specify its name with the **-qsigtrap** option.

If you specify a different handler, ensure that the object module that contains it is linked with the program.

Related Information

The possible causes of exceptions are described in "XL Fortran Run-Time Exceptions" on page 66. "Detecting and Trapping Floating-Point Exceptions" on page 296 describes a number of methods for dealing with exceptions that result from floating-point computations. "Installing an Exception Handler" on page 298 lists the exception handlers that XL Fortran supplies.

-qsmallstack Option

Syntax

-qsmallstack | **-qnosmallstack**

Specifies that the compiler will minimize stack usage where possible.

-qsmp Option

Syntax

-qsmp[=*suboptions*]
-qnosmp

Indicates that code should be produced for an SMP system. The default is to produce code for a uniprocessor machine. When you specify this option, the compiler recognizes all directives with the trigger constants **SMP$**, **$OMP**, and **IBMP** (unless you specify the **omp** suboption).

Only the **xlf_r**, **xlf_r7**, **xlf90_r**, **xlf90_r7**, **xlf95_r**, and **xlf95_r7** invocation commands automatically link in all of the thread-safe components. You can use the **-qsmp** option with the **xlf**, **xlf90**, **xlf95**, **f77**, and **fort77** invocation commands, but you are responsible for linking in the appropriate components. For a description of this, refer to "Linking 32–Bit SMP Object Files Using the ld Command" on page 42. If you use the **-qsmp** option to compile any source file in a program, then you must specify the **-qsmp** option at link time as well, unless you link by using the **ld** command.

Arguments

auto | **noauto**
: This suboption controls automatic parallelization. By default, the compiler will attempt to parallelize explicitly coded **DO** loops as well as those that are generated by the compiler for array language. If you specify the suboption **noauto**, automatic parallelization is turned off, and only constructs that are marked with prescriptive directives are parallelized. If the compiler encounters the **omp** suboption and the **-qsmp** or **-qsmp=auto** suboptions are not explicitly specified on the command line, the **noauto** suboption is implied. Also, note that **-qsmp=noopt** implies **-qsmp=noauto**. No automatic parallelization occurs under **-qsmp=noopt**; only user-directed parallelization will occur.

nested_par | **nonested_par**
: If you specify the **nested_par** suboption, the compiler parallelizes prescriptive nested parallel constructs (**PARALLEL DO**, **PARALLEL SECTIONS**). This includes not only the loop constructs that are nested within a scoping unit but also parallel constructs in subprograms that are referenced (directly or indirectly) from within other parallel constructs. By default, the compiler serializes a nested parallel construct. Note that this option has no effect on loops that are automatically parallelized. In this case, at most one loop in a loop nest (in a scoping unit) will be parallelized.

  Note that the implementation of the **nested_par** suboption does not comply with the OpenMP Fortran API. If you specify this suboption, the run-time library uses the same threads for the nested **PARALLEL DO** and **PARALLEL SECTIONS** constructs that it used for the enclosing **PARALLEL** constructs.

omp | **noomp**
: If you specify **-qsmp=omp**, the compiler enforces compliance with the OpenMP Fortran API. Specifying this option has the following effects:

  - Automatic parallelization is turned off.

- All previously recognized directive triggers are ignored.
- The **-qcclines** compiler option is turned on if you specify **-qsmp=omp.**
- The **-qcclines** compiler option is not turned on if you specify **-qnocclines** and **-qsmp=omp**.
- The only recognized directive trigger is **$OMP**. However, you can specify additional triggers on subsequent **-qdirective** options.
- The compiler issues warning messages if your code contains any language constructs that do not conform to the OpenMP Fortran API.

Specifying this option when the C preprocessor is invoked also defines the **_OPENMP** C preprocessor macro automatically, with the value *200011*, which is useful in supporting conditional compilation. This macro is only defined when the C preprocessor is invoked.

See *Conditional Compilation* in the *Language Elements* section of the *XL Fortran Enterprise Edition for AIX Language Reference* for more information.

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

rec_locks | **norec_locks**
This suboption specifies whether recursive locks are used to avoid problems associated with **CRITICAL** constructs. If you specify the **rec_locks** suboption, a thread can enter a **CRITICAL** construct from within the dynamic extent of another **CRITICAL** construct that has the same name. If you specify **norec_locks**, a deadlock would occur in such a situation.

The default is **norec_locks**, or regular locks.

schedule=*option*
The **schedule** suboption can take any one of the following subsuboptions:

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

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

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

dynamic[=*n*]
The iterations of a loop are divided into chunks containing *n* iterations each. If *n* has not been specified, then the chunks consist of **CEILING**(number_of_iterations / number_of_threads) iterations.

Active threads are assigned these chunks on a ″first-come, first-do″ basis. Chunks of the remaining work are assigned to available threads until all work has been assigned.

If a thread is asleep, its assigned work will be taken over by an active thread once that thread becomes available.

guided[=*n*]
The iterations of a loop are divided into progressively smaller chunks until a minimum chunk size of *n* loop iterations is reached. If *n* has not been specified, the default value for *n* is 1 iteration.

The first chunk contains **CEILING**(number_of_iterations / number_of_threads) iterations. Subsequent chunks consist of **CEILING**(number_of_iterations_left / number_of_threads) iterations. Active threads are assigned chunks on a ″first-come, first-do″ basis.

runtime
Specifies that the chunking algorithm will be determined at run time.

static[=*n*]
The iterations of a loop are divided into chunks containing *n* iterations each. Each thread is assigned chunks in a ″round-robin″ fashion. This is known as *block cyclic scheduling*. If the value of *n* is 1, then the scheduling type is specifically referred to as *cyclic scheduling*.

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

If a thread is asleep and it has been assigned work, it will be awakened so that it may complete its work.

For more information on chunking algorithms and **SCHEDULE**, refer to the *Directives* section in the *XL Fortran Enterprise Edition for AIX Language Reference*.

threshold=*n* Controls the amount of automatic loop parallelization that occurs. The value of *n* represents the lower limit allowed for parallelization of a loop, based on the level of ″work″ present in a loop. Currently, the calculation of ″work″ is weighted heavily by the number of iterations in the loop. In general, the higher the value specified for *n*, the fewer loops are parallelized. If this suboption is not specified, the program will use the default value *n*=100.

Rules

- If you specify **-qsmp** more than once, the previous settings of all suboptions are preserved, unless overridden by the subsequent suboption setting. The compiler does not override previous suboptions that you specify. The same is true for the version of **-qsmp** without suboptions; the default options are saved.
- Specifying the **omp** suboption always implies **noauto**, unless you specify **-qsmp** or **-qsmp=auto** on the command line.
- Specifying the **noomp** suboption always implies **auto**.
- The **omp** and **noomp** suboptions only appear in the compiler listing if you explicitly set them.
- If **-qsmp** is specified without any suboptions, **-qsmp=opt** becomes the default setting. If **-qsmp** is specified after the **-qsmp=noopt** suboption has been set, the **-qsmp=noopt** setting will always be ignored.
- If the option **-qsmp** with no suboptions follows the suboption**-qsmp=noopt** on a command line, the **-qsmp=opt** and **-qsmp=auto** options are enabled.
- Specifying the **-qsmp=noopt** suboption implies that **-qsmp=noauto**. It also implies **-qnoopt**. This option overrides performance options such as **-O2**, **-O3**, **-qhot**, anywhere on the command line unless **-qsmp** appears after **-qsmp=noopt**.
- Object files generated with the **-qsmp=opt** option can be linked with object files generated with **-qsmp=noopt**. The visibility within the debugger of the variables in each object file will not be affected by linking.

Restrictions

The **-qsmp=noopt** suboption may affect the performance of the program.

Within the same **-qsmp** specification, you cannot specify the **omp** suboption before or after certain suboptions. The compiler issues warning messages if you attempt to specify them with **omp**:

auto
: This suboption controls automatic parallelization, but **omp** turns off automatic parallelization.

nested_par
: Note that the implementation of the **nested_par** suboption does not comply with the OpenMP Fortran API. If you specify this suboption, the run-time library uses the same threads for the nested **PARALLEL DO** and **PARALLEL SECTIONS** constructs that it used for the enclosing **PARALLEL** constructs.

rec_locks
: This suboption specifies a behaviour for **CRITICAL** constructs that is inconsistent with the OpenMP Fortran API.

schedule=affinity=*n*
: The affinity scheduling type does not appear in the OpenMP Fortran API standard.

Examples

The **-qsmp=noopt** suboption overrides performance optimization options anywhere on the command line unless **-qsmp** appears after **-qsmp=noopt**. The following examples illustrate that all optimization options that appear after **-qsmp=noopt** are processed according to the normal rules of scope and precedence.

Example 1

```
     xlf90 -qsmp=noopt -O3...
is equivalent to
     xlf90 -qsmp=noopt...
```

Example 2

```
     xlf90 -qsmp=noopt -O3 -qsmp...
is equivalent to
     xlf90 -qsmp -O3...
```

Example 3

```
     xlf90 -qsmp=noopt -O3 -qhot -qsmp -O2...
is equivalent to
     xlf90 -qsmp -qhot -O2...
```

If you specify the following, the compiler recognizes both the **$OMP** and **SMP$** directive triggers and issues a warning if a directive specified with either trigger is not allowed in OpenMP.

```
-qsmp=omp -qdirective=SMP$
```

If you specify the following, the **noauto** suboption is used. The compiler issues a warning message and ignores the **auto** suboption.

```
-qsmp=omp:auto
```

In the following example, you should specify **-qsmp=rec_locks** to avoid a deadlock caused by **CRITICAL** constructs.

```
   program t
     integer  i, a, b

     a = 0
     b = 0
!smp$ parallel do
     do i=1, 10
!smp$ critical
     a = a + 1
!smp$ critical
     b = b + 1
!smp$ end critical
!smp$ end critical
     enddo
   end
```

Related Information

If you use the **xlf**, **xlf_r**, **xlf_r7**, **f77**, or **fort77** command with the **-qsmp** option to compile programs, specify **-qnosave** to make the default storage class automatic, and specify **-qthreaded** to tell the compiler to generate thread-safe code.

-qsource Option

Syntax

```
-qsource | -qnosource
SOURCE | NOSOURCE
```

Determines whether to produce the source section of the listing.

This option displays on the terminal each source line where the compiler detects a problem, which can be very useful in diagnosing program errors in the Fortran source files.

You can selectively print parts of the source code by using **SOURCE** and **NOSOURCE** in **@PROCESS** directives in the source files around those portions of the program you want to print. This is the only situation where the **@PROCESS** directive does not have to be before the first statement of a compilation unit.

Examples

In the following example, the point at which the incorrect call is made is identified more clearly when the program is compiled with the **-qsource** option:

```
$ cat argument_mismatch.f
         subroutine mult(x,y)
         integer x,y
         print *,x*y
         end

         program wrong_args
         interface
                 subroutine mult(a,b)   ! Specify the interface for this
                         integer a,b    ! subroutine so that calls to it
                 end subroutine mult    ! can be checked.
         end interface
         real i,j
         i = 5.0
         j = 6.0
         call mult(i,j)
         end
$ xlf95 argument_mismatch.f
** mult   === End of Compilation 1 ===
"argument_mismatch.f", line 16.12: 1513-061 (S) Actual argument attributes
do not match those specified by an accessible explicit interface.
** wrong_args  === End of Compilation 2 ===
1501-511  Compilation failed for file argument_mismatch.f.
$ xlf95 -qsource argument_mismatch.f
** mult   === End of Compilation 1 ===
        16 |   call mult(i,j)
            ............a...
a - 1513-061 (S) Actual argument attributes do not match those specified by
an accessible explicit interface.
** wrong_args   === End of Compilation 2 ===
1501-511  Compilation failed for file argument_mismatch.f.
```

Related Information

See “Options That Control Listings and Messages” on page 77 and “Source Section” on page 390.

-qspillsize Option

Syntax

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

-qspillsize is the long form of **-NS**. See "-N Option" on page 113.

-qstrict Option

Syntax

```
-qstrict | -qnostrict
STRICT | NOSTRICT
```

Ensures that optimizations done by default with the **-O3**, **-O4**, **-O5**, **-qhot**, and **-qipa** options, and optionally with the **-O2** option, do not alter the semantics of a program.

Defaults

For **-O3**, **-O4**, **-O5**, **-qhot**, and **-qipa**, the default is **-qnostrict**. For **-O2**, the default is **-qstrict**. This option is ignored for **-qnoopt**. With **-qnostrict**, optimizations may rearrange code so that results or exceptions are different from those of unoptimized programs.

This option is intended for situations where the changes in program execution in optimized programs produce different results from unoptimized programs. Such situations are likely rare because they involve relatively little-used rules for IEEE floating-point arithmetic.

Rules

With **-qnostrict** in effect, the following optimizations are turned on, unless **-qstrict** is also specified:

- Code that may cause an exception may be rearranged. The corresponding exception might happen at a different point in execution or might not occur at all. (The compiler still tries to minimize such situations.)
- Floating-point operations may not preserve the sign of a zero value. (To make certain that this sign is preserved, you also need to specify **-qfloat=rrm**, **-qfloat=nomaf**, or **-qfloat=strictnmaf**.)
- Floating-point expressions may be reassociated. For example, **(2.0*3.1)*4.2** might become **2.0*(3.1*4.2)** if that is faster, even though the result might not be identical.
- The **fltint** and **rsqrt** suboptions of the **-qfloat** option are turned on. You can turn them off again by also using the **-qstrict** option or the **nofltint** and **norsqrt** suboptions of **-qfloat**. With lower-level or no optimization specified, these suboptions are turned off by default.

Related Information

See "-O Option" on page 114, "-qhot Option" on page 171, and "-qfloat Option" on page 163.

-qstrictieeemod Option

Syntax

-qstrictieeemod | -qnostrictieeemod
STRICTIEEEMOD | NOSTRICTIEEEMOD

Specifies whether the compiler will adhere to the draft Fortran 2003 IEEE arithmetic rules for the **ieee_arithmetic** and **ieee_exceptions** intrinsic modules. When you specify **-qstrictieeemod**, 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 **–qnostrictieeemod** will relax the rules on saving and restoring floating-point status. This prevents any associated impact on performance.

-qstrict_induction Option

Syntax

-qSTRICT_INDUCtion | **-qNOSTRICT_INDUCtion**

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

You should avoid specifying **-qstrict_induction** unless absolutely necessary, as it may cause performance degradation.

Examples

Consider the following two examples:

Example 1

```
    integer(1) :: i, j                ! Variable i can hold a
    j = 0                             ! maximum value of 127.

    do i = 1, 200                     ! Integer overflow occurs when 128th
      j = j + 1                       ! iteration of loop is attempted.
    enddo
```

Example 2

```
    integer(1) :: i
    i = 1_1                           ! Variable i can hold a maximum
                                      ! value of 127.
100 continue
    if (i == -127) goto 200           ! Go to label 200 once decimal overflow
      i = i + 1_1                     ! occurs and i == -127.
      goto 100
200 continue
    print *, i
    end
```

If you compile these examples with the **-qstrict_induction** option, the compiler does not perform induction variable optimizations, but the performance of the code may be affected. If you compile the examples with the **-qnostrict_induction** option, the compiler may perform optimizations that may alter the semantics of the programs.

-qsuffix Option

Syntax

```
-qsuffix=option=suffix
```

Specifies the source-file suffix on the command line instead of in the **xlf.cfg** file. This option saves time for the user by permitting files to be used as named with minimal makefile modifications and removes the risk of problems associated with modifying the **xlf.cfg** file. Only one setting is supported at any one time for any particular file type.

Arguments

f=*suffix*
: Where *suffix* represents the new *source-file-suffix*

o=*suffix*
: Where *suffix* represents the new *object-file-suffix*

s=*suffix*
: Where *suffix* represents the new *assembler-source-file-suffix*

cpp=*suffix*
: Where *suffix* represents the new *preprocessor-source-file-suffix*

Rules

- The new suffix setting is case-sensitive.
- The new suffix can be of any length.
- Any setting for a new suffix will override the corresponding default setting in the **xlf.cfg** file.
- If both **-qsuffix** and **-F** are specified, **-qsuffix** is processed last, so its setting will override the setting in the **xlf.cfg** file.

Examples

For instance,

```
xlf a.f90 -qsuffix=f=f90:cpp=F90
```

will cause these effects:

- The compiler is invoked for source files with a suffix of .f90.
- cpp is invoked for files with a suffix of .F90.

-qsuppress Option

Syntax

```
-qsuppress[=nnnn-mmm[:nnnn-mmm ...] | cmpmsg]
-qnosuppress
```

Arguments

nnnn-mmm[:nnnn-mmm ...]
: Suppresses the display of a specific compiler message (*nnnn-mmm*) or a list of messages (*nnnn-mmm[:nnnn-mmm ...]*). *nnnn-mmm* is the message number. To suppress a list of messages, separate each message number with a colon.

cmpmsg
: Suppresses the informational messages that report compilation progress and a successful completion.

  This sub-option has no effect on any error messages that are emitted.

Background Information

In some situations, users may receive an overwhelming number of compiler messages. In many cases, these compiler messages contain important information. However, some messages contain information that is either redundant or can be safely ignored. When multiple error or warning messages appear during compilation, it can be very difficult to distinguish which messages should be noted. By using **-qsuppress**, you can eliminate messages that do not interest you.

- The compiler tracks the message numbers specified with **-qsuppress**. If the compiler subsequently generates one of those messages, it will not be displayed or entered into the listing.
- Only compiler and driver messages can be suppressed. Linker or operating system message numbers will be ignored if specified on the **-qextname** compiler option.
- If you are also specifying the **-qipa** compiler option, then **-qipa** must appear before the **-qextname** compiler option on the command line for IPA messages to be suppressed.

Restrictions

- The value of *nnnn* must be a four-digit integer between 1500 and 1585, since this is the range of XL Fortran message numbers.
- The value of *mmm* must be any three-digit integer (with leading zeros if necessary).

Examples

```
@process nullterm
      i = 1; j = 2;
      call printf("i=%d\n",%val(i));
      call printf("i=%d, j=%d\n",%val(i),%val(j));
      end
```

Compiling this sample program would normally result in the following output:

```
"t.f", line 4.36: 1513-029 (W) The number of arguments to "printf" differ
from the number of arguments in a previous reference.  You should use the
OPTIONAL attribute and an explicit interface to define a procedure with
optional arguments.
** _main   === End of Compilation 1 ===
1501-510  Compilation successful for file t.f.
```

When the program is compiled with **-qsuppress=1513-029**, the output is:

```
** _main   === End of Compilation 1 ===
1501-510   Compilation successful for file t.f.
```

Related Information

For another type of message suppression, see "-qflag Option" on page 162.

-qswapomp Option

Syntax

-qswapomp | -qnoswapomp
SWAPOMP | NOSWAPOMP

Specifies that the compiler should recognize and substitute OpenMP routines in XL Fortran programs.

The OpenMP routines for Fortran and C have different interfaces. To support multi-language applications that use OpenMP routines, the compiler needs to recognize OpenMP routine names and substitute them with the XL Fortran versions of these routines, regardless of the existence of other implementations of such routines.

The compiler does not perform substitution of OpenMP routines when you specify the **-qnoswapomp** option.

Restrictions

The **-qswapomp** and **-qnoswapomp** options only affect Fortran sub-programs that reference OpenMP routines that exist in the program.

Rules

- If a call to an OpenMP routine resolves to a dummy procedure, module procedure, an internal procedure, a direct invocation of a procedure itself, or a statement function, the compiler will not perform the substitution.
- When you specify an OpenMP routine, the compiler substitutes the call to a different special routine depending upon the setting of the **-qintsize** option. In this manner, OpenMP routines are treated as generic intrinsic procedures.
- Unlike generic intrinsic procedures, if you specify an OpenMP routine in an **EXTERNAL** statement, the compiler will not treat the name as a user-defined external procedure. Instead, the compiler will still substitute the call to a special routine depending upon the setting of the **-qintsize** option.
- An OpenMP routine cannot be extended or redefined, unlike generic intrinsic procedures.

Examples

In the following example, the OpenMP routines are declared in an **INTERFACE** statement.

```
@PROCESS SWAPOMP

         INTERFACE
            FUNCTION OMP_GET_THREAD_NUM()
                INTEGER OMP_GET_THREAD_NUM
            END FUNCTION OMP_GET_THREAD_NUM

            FUNCTION OMP_GET_NUM_THREADS()
                INTEGER OMP_GET_NUM_THREADS
            END FUNCTION OMP_GET_NUM_THREADS
         END INTERFACE

         IAM = OMP_GET_THREAD_NUM()
         NP = OMP_GET_NUM_THREADS()
         PRINT *, IAM, NP
         END
```

Related Information

See the *OpenMP Execution Environment Routines and Lock Routines* section in the *XL Fortran Enterprise Edition for AIX Language Reference*.

-qtbtable Option

Syntax

```
-qtbtable={none | small | full}
```

Limits the amount of debugging traceback information in object files, which reduces the size of the program.

You can use this option to make your program smaller, at the cost of making it harder to debug. When you reach the production stage and want to produce a program that is as compact as possible, you can specify **-qtbtable=none**. Otherwise, the usual defaults apply: code compiled with **-g** or without **-O** has full traceback information (**-qtbtable=full**), and code compiled with **-O** contains less (**-qtbtable=small**).

Arguments

none The object code contains no traceback information at all. You cannot debug the program, because a debugger or other code-examination tool cannot unwind the program's stack at run time. If the program stops because of a run-time exception, it does not explain where the exception occurred.

small The object code contains traceback information but not the names of procedures or information about procedure parameters. You can debug the program, but some non-essential information is unavailable to the debugger. If the program stops because of a run-time exception, it explains where the exception occurred but reports machine addresses rather than procedure names.

full The object code contains full traceback information. The program is debuggable, and if it stops because of a run-time exception, it produces a traceback listing that includes the names of all procedures in the call chain.

Background Information

This option is most suitable for programs that contain many long procedure names, such as the internal names constructed for module procedures. You may find it more applicable to C++ programs than to Fortran programs.

Restrictions

To use the performance tools, such as **tprof**, in the AIX Performance Toolbox, you must compile the Fortran programs with **-qtbtable=full**.

Related Information

See "-g Option" on page 108, "-O Option" on page 114, "Debugging Optimized Code" on page 321, and "-qcompact Option" on page 142.

-qthreaded Option

Syntax

```
-qthreaded
```

Used by the compiler to determine when it must generate thread-safe code.

The **-qthreaded** option does not imply the **-qnosave** option. The **-qnosave** option specifies a default storage class of automatic for user local variables. In general, both of these options need to be used to generate thread-safe code. Simply specifying these options does not quarantee that your program is thread-safe. You should implement the appropriate locking mechanisms, as well.

Defaults

-qthreaded is the default for the **xlf90_r**, **xlf90_r7**, **xlf95_r**, **xlf95_r7**, **xlf_r**, and **xlf_r7** commands.

Specifying the **-qthreaded** option implies **-qdirective=ibmt**, and by default, the *trigger_constant* **IBMT** is recognized.

-qtune Option

Syntax

`-qtune=`*`implementation`*

Tunes instruction selection, scheduling, and other implementation-dependent performance enhancements for a specific implementation of a hardware architecture. The compiler will use a **-qtune** setting that is compatible with the target architecture, which is controlled by the **-qarch**, **-q32**, and **-q64** options.

If you want your program to run on more than one architecture, but to be tuned to a particular architecture, you can use a combination of the **-qarch** and **-qtune** options. These options are primarily of benefit for floating-point intensive programs.

By arranging (scheduling) the generated machine instructions to take maximum advantage of hardware features such as cache size and pipelining, **-qtune** can improve performance. It only has an effect when used in combination with options that enable optimization.

Although changing the **-qtune** setting may affect the performance of the resulting executable, it has no effect on whether the executable can be executed correctly on a particular hardware platform.

Arguments

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

pwr
: The optimizations are tuned for the POWER processors.

pwr2
: The optimizations are tuned for the POWER2 processors.

: **pwrx** is a synonym for **pwr2**, but **pwr2** is preferred.

pwr2s
: The optimizations are tuned for the desktop implementation of the POWER2 architecture, which has a narrower processor-to-memory bus than other POWER2 implementations. Quad-word instructions, which are slower on these machines than on other POWER2 machines, are deemphasized to reduce bus contention. That is, there may be fewer of them or none at all.

p2sc
: The optimizations are tuned for the POWER2 Super Chip.

601
: The optimizations are tuned for the PowerPC 601 processor.

603
: The optimizations are tuned for the PowerPC 603 processor.

604
: The optimizations are tuned for the PowerPC 604 processor.

rs64a
: The optimizations are tuned for the RS64I processor.

rs64b
: The optimizations are tuned for the RS64II processor.

rs64c
: The optimizations are tuned for the RS64III processor.

pwr3
: The optimizations are tuned for the POWER3 processors.

pwr4
: The optimizations are tuned for the POWER4 processors.

pwr5
: The optimizations are tuned for the POWER5 processors.

ppc970
: The optimizations are tuned for the PowerPC 970 processors.

If you do not specify **-qtune**, its setting is determined by the setting of the **-qarch** option, as follows:

-qarch Setting	Allowed -qtune Settings	Default -qtune Setting
com (if you specify -q32)	pwr, pwr2/pwrx, pwr3, pwr4, pwr2s, p2sc, rs64a, rs64b, rs64c, 601, 603, 604, auto	pwr2 (if you specify -q32)
com (if you specify -q64)	See the list of acceptable **-qtune** settings under the **-qarch=ppc64** entry.	pwr3 (if you specify -q64)
pwr	pwr, pwr2/pwrx, pwr2s, p2sc, 601, auto	pwr2
pwr2/pwrx	pwr2/pwrx, p2sc, pwr2s	pwr2/pwrx
pwr2s	pwr2s, auto	pwr2s
p2sc	p2sc, auto	p2sc
601	601, auto	601
603	603, auto	603
604	604, auto	604
ppc (if you specify -q32)	601, 603, 604, rs64a, rs64b, rs64c, pwr3, pwr4, pwr5, ppc970, auto	pwr3 (if you specify -q32)
ppc (if you specify -q64)	See the list of acceptable **-qtune** settings under the **-qarch=ppc64** entry.	pwr3 (if you specify -q64)
ppcgr (if you specify -q32)	603, 604, rs64b, rs64c, pwr3, pwr4, pwr5, ppc970, auto	pwr3 (if you specify -q32)
ppcgr (if you specify -q64)	See the list of acceptable **-qtune** settings under the **-qarch=ppc64gr** entry.	pwr3 (if you specify -q64)
ppc64	rs64a, rs64b, rs64c, pwr3, pwr4, pwr5, ppc970, auto	pwr3
ppc64gr	rs64b, rs64c, pwr3, pwr4, pwr5, ppc970, auto	pwr3
ppc64grsq	rs64b, rs64c, pwr3, pwr4, pwr5, ppc970, auto	pwr3
rs64a	rs64a, auto	rs64a
rs64b	rs64b, auto	rs64b
rs64c	rs64c, auto	rs64c
pwr3	pwr3, pwr4, pwr5, ppc970, auto	pwr3
pwr4	pwr4, pwr5, ppc970, auto	pwr4
pwr5	pwr5, auto	pwr5
ppc970	ppc970, auto	ppc970

Note that you can specify any **-qtune** suboption with **-qarch=auto** as long as you are compiling on a machine that is compatible with the **-qtune** suboption. For example, if you specify **-qarch=auto** and **-qtune=pwr5**, you must compile on a POWER3, POWER4, or POWER5 machine.

Restrictions

Because reducing quad-word instructions may degrade performance on other POWER2 models, we do not recommend the **pwr2s** suboption for programs that will be run on a number of different POWER2 models. If the program will be run on a set of different POWER2 models, leave the **-qtune** setting as **pwr2**.

Related Information

See “-qarch Option” on page 127, “-qcache Option” on page 137, and “Compiling for Specific Architectures” on page 39.

-qundef Option

Syntax

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

-qundef is the long form of the "-u Option" on page 272.

-qunroll Option

Syntax

-qunroll[=**auto** | yes] | -qnounroll

Specifies whether unrolling a **DO** loop is allowed in a program. Unrolling is allowed on outer and inner **DO** loops.

Arguments

auto
: The compiler performs basic loop unrolling. This is the default if **-qunroll** is not specified on the command line.

yes
: The compiler looks for more opportunities to perform loop unrolling than that performed with **-qunroll=auto**. Specifying **-qunroll** with no suboptions is equivalent to **-qunroll=yes**. In general, this suboption has more chances to increase compile time or program size than **-qunroll=auto** processing, but it may also improve your application's performance.

If you decide to unroll a loop, specifying one of the above suboptions does not automatically guarantee that the compiler will perform the operation. Based on the performance benefit, the compiler will determine whether unrolling will be beneficial to the program. Experienced compiler users should be able to determine the benefit in advance.

Rules

The **-qnounroll** option prohibits unrolling unless you specify the **STREAM_UNROLL**, **UNROLL**, or **UNROLL_AND_FUSE** directive for a particular loop. These directives always override the command line options.

Examples

In the following example, the **UNROLL(2)** directive is used to tell the compiler that the body of the loop can be replicated so that the work of two iterations is performed in a single iteration. Instead of performing 1000 iterations, if the compiler unrolls the loop, it will only perform 500 iterations.

```
!IBM* UNROLL(2)
      DO I = 1, 1000
         A(I) = I
      END DO
```

If the compiler chooses to unroll the previous loop, the compiler translates the loop so that it is essentially equivalent to the following:

```
      DO I = 1, 1000, 2
         A(I) = I
         A(I+1) = I + 1
      END DO
```

Related Information

See the appropriate directive on unrolling loops in the *XL Fortran Enterprise Edition for AIX Language Reference*.

- **STREAM_UNROLL**
- **UNROLL**
- **UNROLL_AND_FUSE**

See "Optimizing Loops and Array Language" on page 312.

-qunwind Option

Syntax

```
-qunwind |-qnounwind
 UNWIND  | NOUNWIND
```

Specifies that the compiler will preserve the default behavior for saves and restores to volatile registers during a procedure call. If you specify **-qnounwind**, the compiler rearranges subprograms to minimize saves and restores to volatile registers.

While code semantics are preserved, applications such as exception handlers that rely on the default behavior for saves and restores can produce undefined results. When using **-qnounwind** in conjunction with the **-g** compiler option, debug information regarding exception handling when unwinding the program's stack can be inaccurate.

-qversion Option

Syntax

```
-qversion | -qnoversion
```

Displays the version and release of the invoking compiler. Specify this option on its own with the compiler command. For example:

```
xlf90 -qversion
```

-qwarn64 Option

See "-qwarn64 Option" on page 284.

-qxflag=oldtab Option

Syntax

```
-qxflag=oldtab
XFLAG(OLDTAB)
```

Interprets a tab in columns 1 to 5 as a single character (for fixed source form programs), for compatibility with XL Fortran Version 1.

Defaults

By default, the compiler allows 66 significant characters on a source line after column 6. A tab in columns 1 through 5 is interpreted as the appropriate number of blanks to move the column counter past column 6. This default is convenient for those who follow the earlier Fortran practice of including line numbers or other data in columns 73 through 80.

Rules

If you specify the option **-qxflag=oldtab**, the source statement still starts immediately after the tab, but the tab character is treated as a single character for counting columns. This setting allows up to 71 characters of input, depending on where the tab character occurs.

-qxflag=xalias Option

Syntax

```
-qxflag=xalias
XFLAG(XALIAS)
```

Obsolete: replaced by **-qalias=nostd**. See "-qalias Option" on page 122 instead.

-qxlf77 Option

Syntax

```
-qxlf77=settings
XLF77(settings)
```

Provides backward compatibility with XL Fortran for AIX Versions 1 and 2 aspects of language semantics and I/O data format that have changed. Most of these changes are required by the Fortran 90 standard.

Defaults

By default, the compiler uses settings that apply to Fortran 95, Fortran 90, and the most recent compiler version in all cases; the default suboptions are **blankpad**, **nogedit77**, **nointarg**, **nointxor**, **leadzero**, **nooldboz**, **nopersistent**, and **nosofteof**. However, these defaults are only used by the **xlf95**, **xlf95_r**, **xlf95_r7**, **xlf90**, **xlf90_r**, **xlf90_r7**, **f90**, and **f95** commands, which you should use to compile new programs.

For maximum compatibility for programs and data created for XL Fortran Versions 1 and 2, the **xlf**, **xlf_r**, **xlf_r7**, **f77**, and **fort77** commands use the opposite settings for this option.

If you only want to compile and run old programs unchanged, you can continue to use the appropriate invocation command and not concern yourself with this option. You should only use this option if you are using existing source or data files with Fortran 90 or Fortran 95 and the **xlf90**, **xlf90_r**, **xlf90_r7**, **xlf95**, **xlf95_r**, **xlf95_r7**, **f90**, or **f95** command and find some incompatibility because of behavior or data format that has changed since XL Fortran Version 2. Eventually, you should be able to recreate the data files or modify the source files to remove the dependency on the old behavior.

Arguments

To get various aspects of XL Fortran Version 2 behavior, select the nondefault choice for one or more of the following suboptions. The descriptions explain what happens when you specify the nondefault choices.

blankpad | **noblankpad**
: For internal , direct-access, and stream-access files, uses a default setting equivalent to **pad='no'**. This setting produces conversion errors when reading from such a file if the format requires more characters than the record has, thus duplicating the XL Fortran Version 2 behavior. 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, for compatibility with XL Fortran Version 2. 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.

Related Information

See "Avoiding or Fixing Upgrade Problems" on page 25.

-qxlf90 Option

Syntax

```
-qxlf90={settings}
XLF90({settings})
```

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

Defaults

The default suboptions for **-qxlf90** depend on the invocation command that you specify. For the **xlf95**, **xlf95_r** , or **xlf95_r7** command, the default suboptions are **signedzero** and **autodealloc**. For all other invocation commands, the defaults are **nosignedzero** and **noautodealloc**.

Arguments

signedzero | **nosignedzero**
: Determines how the **SIGN(A,B)** function handles signed real 0.0. Prior to XL Fortran Version 6.1, **SIGN(A,B)** returned |A| when B=-0.0. This behavior conformed with the Fortran 90 standard. Now, if you specify the **-qxlf90=signedzero** compiler option, **SIGN(A,B)** returns -|A| when B=-0.0. This behavior conforms to the Fortran 95 standard and is consistent with the IEEE standard for binary floating-point arithmetic. Note that for the **REAL(16)** data type, XL Fortran never treats zero as negative zero.

  This suboption also determines whether a minus sign is printed in the following cases:

  - For a negative zero in formatted output. Again, note that for the **REAL(16)** data type, XL Fortran never treats zero as negative zero.
  - For negative values that have an output form of zero (that is, where trailing non-zero digits are truncated from the output so that the resulting output looks like zero). Note that in this case, the **signedzero** suboption does affect the **REAL(16)** data type; non-zero negative values that have an output form of zero will be printed with a minus sign.

autodealloc | **noautodealloc**
: Determines whether the compiler deallocates allocatable objects that are declared locally without either the **SAVE** or the **STATIC** attribute and have a status of currently allocated when the subprogram terminates. This behavior conforms with the Fortran 95 standard and did not exist in XL Fortran prior to Version 6.1. If you are certain that you are deallocating all local allocatable objects explicitly, you may wish to turn off this suboption to avoid possible performance degradation.

Examples

Consider the following program:

```
PROGRAM TESTSIGN
REAL X, Y, Z
X=1.0
Y=-0.0
Z=SIGN(X,Y)
PRINT *,Z
END PROGRAM TESTSIGN
```

The output from this example depends on the invocation command and the **-qxlf90** suboption that you specify. For example:

Invocation Command/xlf90 Suboption	Output
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 *Arrays Concepts* section of the *XL Fortran Enterprise Edition for AIX Language Reference*.

-qxlines Option

Syntax

```
-qxlines | -qnoxlines
XLINES |  NOXLINES
```

Specifies whether fixed source form lines with a X in column 1 are compiled or treated as comments. This option is similar to the recognition of the character 'd' in column 1 as a conditional compilation (debug) character. The **-D** option recognizes the character 'x' in column 1 as a conditional compilation character when this compiler option is enabled. The 'x' in column 1 is interpreted as a blank, and the line is handled as source code.

Defaults

This option is set to **-qnoxlines** by default, and lines with the character 'x' in column 1 in fixed source form are treated as comment lines. While the **-qxlines** option is independent of **-D**, all rules for debug lines that apply to using 'd' as the conditional compilation character also apply to the conditional compilation character 'x'. The **-qxlines** compiler option is only applicable to fixed source form.

The conditional compilation characters 'x' and 'd' may be mixed both within a fixed source form program and within a continued source line. If a conditional compilation line is continued onto the next line, all the continuation lines must have 'x' or 'd' in column 1. If the initial line of a continued compilation statement is not a debugging line that begins with either 'x' or 'd' in column 1, subsequent continuation lines may be designated as debug lines as long as the statement is syntactically correct.

The OMP conditional compilation characters '!$', 'C$', and '*$' may be mixed with the conditional characters 'x' and 'd' both in fixed source form and within a continued source line. The rules for OMP conditional characters will still apply in this instance.

Examples

An example of a base case of -qxlines:

```
      C2345678901234567890
             program p
             i=3 ; j=4 ; k=5
      X      print *,i,j
      X    +        ,k
             end program p

     <output>: 3 4 5     (if -qxlines is on)
               no output (if -qxlines is off)
```

In this example, conditional compilation characters 'x' and 'd' are mixed, with 'x' on the initial line:

```
C234567890123456789 0
       program p
       i=3 ; j=4 ; k=5
X      print *,i,
D    +         j,
X    +         k
       end program p

<output>: 3 4 5 (if both -qxlines and -qdlines are on)
          3 5   (if only -qxlines is turned on)
```

Here, conditional compilation characters 'x' and 'd' are mixed, with 'd' on the initial line:

```
C2345678901234567890
       program p
       i=3 ; j=4 ; k=5
D      print *,i,
X    +         j,
D    +         k
       end program p

<output>: 3 4 5 (if both -qxlines and -qdlines are on)
          3 5   (if only -qdlines is turned on)
```

In this example, the initial line is not a debug line, but the continuation line is interpreted as such, since it has an 'x' in column 1:

```
C2345678901234567890
       program p
       i=3 ; j=4 ; k=5
       print *,i
X    +        ,j
X    +        ,k
       end program p

<output>: 3 4 5 (if -qxlines is on)
          3     (if -qxlines is off)
```

Related Information

See "-D Option" on page 105 and *Conditional Compilation* in the *Language Elements* section of the *XL Fortran Enterprise Edition for AIX Language Reference*.

-qxref Option

Syntax

```
-qxref[=full] | -qnoxref
XREF[(FULL)] | NOXREF
```

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

If you specify only **-qxref**, only identifiers that are used are reported. If you specify **-qxref=full**, the listing contains information about all identifiers that appear in the program, whether they are used or not.

If **-qxref** is specified after **-qxref=full**, the full cross-reference listing is still produced.

You can use the cross-reference listing during debugging to locate problems such as using a variable before defining it or entering the wrong name for a variable.

Related Information

See "Options That Control Listings and Messages" on page 77 and "Attribute and Cross-Reference Section" on page 392.

-qzerosize Option

Syntax

```
-qzerosize | -qnozerosize
ZEROSIZE | NOZEROSIZE
```

Improves performance of FORTRAN 77 and some Fortran 90 and Fortran 95 programs by preventing checking for zero-sized character strings and arrays.

For Fortran 90 and Fortran 95 programs that might process such objects, use **-qzerosize**. For FORTRAN 77 programs, where zero-sized objects are not allowed, or for Fortran 90 and Fortran 95 programs that do not use them, compiling with **-qnozerosize** can improve the performance of some array or character-string operations.

Defaults

The default setting depends on which command invokes the compiler: **-qzerosize** for the **xlf90**, **xlf90_r**, **xlf90_r7**, **xlf95**, **xlf95_r**, **xlf95_r7**, **f90**, and **f95** commands and **-qnozerosize** for the **xlf**, **xlf_r**, **xlf_r7**, and **f77**/**fort77** commands (for compatibility with FORTRAN 77).

Rules

Run-time checking performed by the **-C** option takes slightly longer when **-qzerosize** is in effect.

-S Option

Syntax

```
-S
```

Produces one or more **.s** files that show equivalent assembler source for each Fortran source file.

Rules

When this option is specified, the compiler produces the assembler source files as output instead of an object or an executable file.

Restrictions

The generated assembler files do not include all the data that is included in a **.o** file by **-qipa** or **-g**.

Examples

```
xlf95 -O3 -qhot -S test.f              # Produces test.s
```

Related Information

The "-o Option" on page 116 can be used to specify a name for the resulting assembler source file.

For information about the assembler-language format, see the *Assembler Language Reference*.

-t Option

Syntax

-tcomponents

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

Rules

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

Component	-t Mnemonic	Standard Program Name
assembler	a	as
C preprocessor	F	cpp
VAST-2 preprocessor	p	fpp
KAP preprocessor	p	fppk
compiler front end	c	xlfentry
array language optimizer	h	xlfhot
IPA/loop optimizer	I	ipa
code generator	b	xlfcode
binder	z	bolt
linker	l	ld
disassembler	d	dis

Related Information

See "-B Option" on page 93 (which includes an example).

-U Option

Syntax

```
-U
MIXED | NOMIXED
```

Makes the compiler sensitive to the case of letters in names.

You can use this option when writing mixed-language programs, because Fortran names are all lowercase by default, while names in C and other languages may be mixed-case.

Rules

If **-U** is specified, case is significant in names. For example, the names `Abc` and `ABC` refer to different objects.

The option changes the link names used to resolve calls between compilation units. It also affects the names of modules and thus the names of their **.mod** files.

Defaults

By default, the compiler interprets all names as if they were in lowercase. For example, `Abc` and `ABC` are both interpreted as `abc` and so refer to the same object.

Restrictions

The names of intrinsics must be all in lowercase when **-U** is in effect. Otherwise, the compiler may accept the names without errors, but the compiler considers them to be the names of external procedures, rather than intrinsics.

The XL Fortran Version 2 requirement that keywords be all lowercase no longer applies.

Related Information

This is the short form of **-qmixed**. See “-qmixed Option” on page 202.

-u Option

Syntax

```
-u
UNDEF | NOUNDEF
```

Specifies that no implicit typing of variable names is permitted. It has the same effect as using the **IMPLICIT NONE** statement in each scope that allows implicit statements.

Defaults

By default, implicit typing is allowed.

Related Information

See **IMPLICIT** in the *XL Fortran Enterprise Edition for AIX Language Reference*.

This is the short form of **-qundef**. See "-qundef Option" on page 254.

-v Option

Syntax

```
-v
```

Generates information on the progress of the compilation.

Rules

As the compiler executes commands to perform different compilation steps, this option displays a simulation of the commands it calls and the system argument lists it passes.

For a particular compilation, examining the output that this option produces can help you determine:

- What files are involved
- What options are in effect for each step
- How far a compilation gets when it fails

Related Information

"-# Option" on page 91 is similar to **-v**, but it does not actually execute any of the compilation steps.

-V Option

Syntax

```
-V
```

This option is the same as **-v** except that you can cut and paste directly from the display to create a command.

-W Option

Syntax

`-Wcomponent,options`

Passes the listed options to a component that is executed during compilation. *component* is one of **a**, **F**, **p**, **c**, **h**, **I**, **b**, **z**, **l**, or **d**, corresponding to the assembler, the C preprocessor, an optimizing preprocessor, the compiler, the array language optimizer, the interprocedural analysis (IPA) tool/loop optimizer, the code generator, the binder, the linker, and the **-S** disassembler, respectively.

Component	-W Mnemonic	Standard Program Name
assembler	a	as
C preprocessor	F	cpp
VAST-2 preprocessor	p	fpp
KAP preprocessor	p	fppk
compiler front end	c	xlfentry
array language optimizer	h	xlfhot
IPA/loop optimizer	I	ipa
code generator	b	xlfcode
binder	z	bolt
linker	l	ld
disassembler	d	dis

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

```
-Wcomponent,option_1[option_2,...,option_n]
```

Background Information

The primary purpose of this option is to construct sequences of compiler options to pass to one of the optimizing preprocessors. It can also be used to fine-tune the link-edit step by passing parameters to the **ld** command.

Defaults

You do not need the **-W** option to pass most options to the linker: unrecognized command-line options, except **-q** options, are passed to it automatically. Only linker options with the same letters as compiler options, such as **-v** or **-S**, strictly require **-W** (or the **ldopts** stanza in the configuration file).

If you need to include a character that is special to the shell in the option string, precede the character with a backslash.

Examples

See “Passing Command-Line Options to the ″ld″ or ″as″ Commands” on page 38.

You can use \, to embed a literal comma in the string supplied to the **-W** option.

In the following example, the \, embeds a literal comma in the **-WF** string and causes three arguments, rather than four, to be supplied to the C preprocessor.

```
$ xlf -qfree=f90  '-WF,-Dint1=1,-Dint2=2,-Dlist=3\,4'  a.F
$ cat a.F
print *, int1
print *, int2
print *, list
end
```

The output from the program will be:

```
$ ./a.out
 1
 2
 3 4
```

-w Option

Syntax

```
-w
```

A synonym for the "-qflag Option" on page 162. It sets **-qflag=e:e**, suppressing warning and informational messages and also messages generated by language-level checking.

-y Option

Syntax

```
-y{n | m | p | z}
IEEE(Near | Minus | Plus | Zero)
```

Specifies the rounding mode for the compiler to use when evaluating constant floating-point expressions at compile time. It is equivalent to the **-qieee** option.

Arguments

n Round to nearest.

m Round toward minus infinity.

p Round toward plus infinity.

z Round toward zero.

Related Information

See "-O Option" on page 114 and "-qfloat Option" on page 163.

-y is the short form of "-qieee Option" on page 175.

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 AIX operating system provides an environment that allows you to develop and execute programs that exploit 64-bit processors through the use of 64-bit pointers and 64-bit integers. XL Fortran only supports applications that target the 64-bit Large Data Type (LDT) Application Binary Interface (ABI). Non-LDT environments, such as AIX version 4, are no longer supported.

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 binder binds 64-bit objects to create 64-bit executables. Note that objects that are bound together, statically or shared, must all be of the same object format. The following scenarios are not permitted and will fail to load, or execute, or both:

- A 64-bit object or executable that has references to symbols from a 32-bit library or shared library
- A 32-bit object or executable that has references to symbols from a 64-bit library or shared library
- A 64-bit executable that attempts to explicitly load a 32-bit module
- A 32-bit executable that attempts to explicitly load a 64-bit module
- Attempts to run 64-bit applications on 32-bit platforms

On both 64-bit and 32-bit platforms, 32-bit executables will continue to run as they currently do on a 32-bit platform.

The XL Fortran compiler mainly provides 64-bit mode support through the compiler option **-q64** in conjunction with the compiler option **-qarch**. This combination determines the bit mode and instruction set for the target architecture. The **-q32** and **-q64** options take precedence over the setting of the **-qarch** option. The **-q64** option will win over a 32-bit mode only **-qarch** setting, and the compiler will upgrade the **-qarch** setting to something that will handle 64-bit mode. Conflicts between the **-q32** and **-q64** options are resolved by the ″last option wins″ rule. Setting **-qarch=com** will ensure future compatibility for applications in 32–bit mode. For 64-bit mode applications, use **-qarch=ppc64** to achieve the same effect for all present or future supported 64-bit mode systems. **-qarch** settings that target a specific architecture, like the **603**, **604**, **rs64a**, **rs64b**, **rs64c**, **pwr3**, **pwr4**, **pwr5**, **ppc970**, and **auto** settings will be more system-dependent.

64-Bit Large Data Type Support

The 64-bit Large Data Type (LDT) Application Binary Interface (ABI), or 64-bit LDT ABI, increases scalability for 64-bit applications, while maintaining binary compatibility for existing 32-bit applications. To accomplish this, some system-derived types are increased from 32-bits to 64-bits. In addition, a new 64-bit magic number is being introduced in the XCOFF definition to identify object code files using the new 64-bit ABI.

The AIX 4.3 64-bit, non-LDT, ABI is no longer supported on AIX 5.1. Object code files with the old 64-bit magic number will not link, load, or execute. Pre-AIX 5.1 64-bit applications must be recomplied in order to execute them on AIX 5.1. Binary compatibility will be preserved for 32-bit objects generated on all earlier levels of AIX, regardless of LDT support.

64-Bit Thread Support

On AIX Version 5.1 with the POSIX 1003.1-1996 standard pthreads API, XL Fortran , beginning with Version 5.1.1, supports 64-bit thread programming. You can specify the **-q64** compiler option with the **xlf_r**, **xlf_r7**, **xlf90_r**, **xlf90_r7**, **xlf95_r**, and **xlf95_r7** commands. For example, you can specify the following command to compile and then link a program in 64-bit object mode:

```
xlf90_r -q64 -qsmp test.f
```

AIX Version 5.1 supports the POSIX 1003.1-1996 standard interface in both 32-bit and 64-bit object mode, but supports the Draft 7 interface in 32-bit object mode only. That is, the **libpthreads.a** library has a 32-bit and a 64-bit part, while the **libpthreads_compat.a** and **libxlfpthrds_compat.a** libraries have 32-bit parts only.

Compiler Options for the 64-Bit Environment

The compiler options that are described in this section enable you to do the following:
- Develop applications for the 64-bit environment
- Help migrate source code from the 32-bit environment to a 64-bit environment

Some of these options already exist in the 32-bit environment but have new settings particular to the 64-bit architecture. This section only covers the new settings for these cases. The options that are grouped here are primarily for developers who are targetting 64-bit platforms.

-q32 Option

Syntax

```
-q32
```

Enables 32-bit compilation bit mode (or, more briefly, 32-bit mode) support in a 64-bit environment. The **-q32** option indicates the compilation bit mode and, together with the **-qarch** option, determines the target machines that the 32-bit executable will run on.

Rules

- The default integer and default real size are 4 bytes in 32-bit mode.
- The default integer pointer size is 4 bytes in 32-bit mode.
- 32-bit object modules are created when targeting 32-bit mode.
- **-q32** is the default, if you have not specified either **-q32** or **-q64** and if you have not set the **OBJECT_MODE** environment variable. For a description of the **OBJECT_MODE** environment variable, see "Default Bit Mode" on page 285.
- **-q64** may override **-q32**.
- All settings for **-qarch** are compatible with **-q32**. If you specify **-q32**, the default **-qarch** suboption is **com**, and the default **-qtune** suboption for **-q32** is **pwr2**.
- The **LOC** intrinsic returns an **INTEGER(4)** value.

Examples

- Using 32-bit compilation mode and targetting a generic PowerPC architecture:

  ```
  -qarch=ppc -q32
  ```
- Now keep the same compilation mode, but change the target to `RS64II`:

  ```
  -qarch=ppc -q32 -qarch=rs64b
  ```

  Notice that the last setting for **-qarch** wins.
- Now keep the same target, but change the compilation mode to 64-bit:

  ```
  -qarch=ppc -q32 -qarch=rs64b -q64
  ```

  Notice that specifying **-q64** overrides the earlier instance of **-q32**.

-q64 Option

Syntax

```
-q64[=largetype]
```

Indicates the 64-bit compilation bit mode and, together with the **-qarch** option, determines the target machines on which the 64-bit executable will run. The **-q64** option indicates that the object module will be created in 64-bit object format and that the 64-bit instruction set will be generated. Note that you may compile in a 32-bit environment to create 64-bit objects, but you must link them in a 64-bit environment with the **-q64** option.

Defaults

-q64=largetype is always true when compiling with **-q64** on AIX 5.1 and above. The **-q64=largetype** suboption is for compatiblity with older XL Fortran compilers. It is not necessary to specifiy it to generate the 64-bit LDT ABI. **-q64=nolargetype** is no longer a supported compiler option and will generate a warning if used.

Rules

- Settings for **-qarch** that are compatible with **-q64** are as follows:
  - **-qarch=auto** (if compiling on a 64-bit system)
  - **-qarch=com** (With **-q64** and **-qarch=com**, the compiler will silently upgrade the arch setting to **ppc64**.)
  - **-qarch=ppc** (With **-q64** and **-qarch=ppc**, the compiler will silently upgrade the arch to **ppc64**.)
  - **-qarch=ppcgr** (With **-q64** and **-qarch=ppcgr**, the compiler will silently upgrade the arch to **ppc64gr**.)
  - **-qarch=ppc64**
  - **-qarch=ppc64gr**
  - **-qarch=ppc64grsq**
  - **-qarch=rs64a**
  - **-qarch=rs64b**
  - **-qarch=rs64c**
  - **-qarch=pwr3**
  - **-qarch=pwr4**
  - **-qarch=pwr5**
  - **-qarch=ppc970**
- The default **-qarch** setting for **-q64** is **ppc64**.
- 64-bit object modules are created when targeting 64-bit mode.
- **-q32** may override **-q64**.
- **-q64** will override a conflicting setting for **-qarch**.

  For example,

  ```
  -q64 -qarch=604
  ```

  will be changed to the setting **-qarch=ppc64**.
- The default tune setting for **-q64** is **-qtune=pwr3**.
- The default integer and default real size is 4 bytes in 64-bit mode.
- The default integer pointer size is 8 bytes in 64-bit mode.
- The maximum array size increases to approximately 2**40 bytes (in static storage) or 2**60 bytes (in dynamic allocation on the heap). The maximum dimension bound range is extended to -2**63, 2**63-1 bytes. The maximum array size for array constants has not been extended and will remain the same as the maximum in 32-bit mode. The maximum array size that you can initialize is 2**28 bytes.
- The maximum iteration count for array constructor implied DO loops increases to 2**63-1 bytes.

- The maximum character variable length extends to approximately 2**40 bytes. The maximum length of character constants and subobjects of constants remains the same as in 32-bit mode, which is 32 767 bytes (32 KB).
- The **LOC** intrinsic returns an **INTEGER(8)** value.
- When you use **-qautodbl=dblpad** in 64-bit mode, you should use **-qintsize=8** to promote **INTEGER(4)** to **INTEGER(8)** for 8 byte integer arithmetic.

Restrictions

- Objects that are generated with the 64-bit LDT ABI are not compatible with 64-bit non-LDT ABI objects. 64-bit LDT ABI objects cannot be linked with 64-bit non-LDT ABI objects on any level of AIX. XL Fortran no longer supports non-LDT ABI objects either for producing them or linking with them.
- 64-bit LDT ABI objects must be linked on AIX 5.1 and above. Objects created with the old 64-bit non-LDT ABI must be linked on AIX 4.3.3. XL Fortran no longer supports emitting code that executes on AIX version 4.
- 64-bit LDT ABI applications cannot be loaded or executed on AIX 4.3.3. 64-bit non-LDT ABI applications cannot be loaded or executed on AIX 5.1 and above. **Existing 64-bit applications must be recompiled to run on AIX 5.1 and above!**
- 64-bit module (.mod) files created by previous versions of XL Fortran can only be used if they were compiled with **-q64=largetype**.
- The compiler no longer appends **ldt** to the specified directory name at compilation time with **-q64=largetype** if the **include_64** attribute is used in the **xlf.cfg** file to specify an alternate directory for 64-bit include and module files. As a result, you may need to change your existing build environment.
- Each of the following situations will produce an error message:
  1. Attempting to link, load, or execute conflicting 64-bit ABI objects on any level of AIX
  2. Attempting to link, load, or execute 64-bit LDT ABI objects on AIX 4.3.3
  3. Attempting to link, load, or execute 64-bit non-LDT ABI objects on AIX 5.1 and above.

Examples

This example targets the RS64II (also known as RS64b) in 64-bit mode:

```
-q32 -qarch=rs64b -q64
```

In this example 64-bit compilation that targets the common group of 64-bit architectures (which currently consists only of the RS64I,RS64II, RS64III, POWER3, POWER4, POWER5, and PowerPC 970):

```
-q64 -qarch=com
```

arch setting is silently upgraded to **ppc64**, the most ″common″ 64-bit mode compilation target.

In this example, the **-qarch** option conflicts with **-q64**:

```
-qarch=604 -q64
```

which results in a suboption setting of **-q64 -qarch=ppc64** and a warning message.

In the example that follows, the **-qarch** option conflicts with **-q64**:

```
-q64 -qarch=604
```

which results in a suboption setting of **-q64 -qarch=ppc64** and a warning message.

-qwarn64 Option

Syntax

`-qwarn64 | -qnowarn64`

Aids in porting code from a 32-bit environment to a 64-bit environment by detecting the truncation of an 8-byte integer pointer to 4 bytes. The **-qwarn64** option uses informational messages to identify statements that may cause problems with the 32-bit to 64-bit migration.

Rules

- The default setting is **-qnowarn64**.
- You can use the **-qwarn64** option in both 32-bit and 64-bit modes.
- The compiler flags the following situations with informational messages:
  - The assignment of a reference to the **LOC** intrinsic to an **INTEGER(4)** variable.
  - The assignment between an **INTEGER(4)** variable or **INTEGER(4)** constant and an integer pointer.
  - The specification of an integer pointer within a common block. We recommend the **-qextchk** option for common block length changes.
  - The specification of an integer pointer within an equivalence statement.
- We additionally recommend the **-qextchk** option and interface blocks for argument checking.

Default Bit Mode

The AIX operating system provides support for the **OBJECT_MODE** environment variable to enable the user to obtain a 64-bit development environment. AIX tools use the setting of **OBJECT_MODE** to determine the type of object to be used or created. The **OBJECT_MODE** environment variable has three recognized settings:

OBJECT_MODE=32
Works with 32-bit objects

OBJECT_MODE=64
Works with 64-bit objects

OBJECT_MODE=32_64
Works with either 32-bit or 64-bit objects

The XL Fortran compiler determines the default bit mode through the setting of the **OBJECT_MODE** environment variable at the time of invocation. The following table shows the default bit mode and options that are set for each setting of the **OBJECT_MODE** environment variable:

Table 16. Default bit mode determined by the setting of OBJECT_MODE

OBJECT_MODE Setting	Default Bit Mode	Default Option Set
unset	32-bit	-q32
32	32-bit	-q32
64	64-bit	-q64
32_64	Not permitted	n/a

Specification of the following options on the command line or in the configuration file overrides the default option set:
- **-q64**
- **-q32**

Important Note

Using **OBJECT_MODE** to determine the default bit mode can have serious implications if you are not aware of the setting of **OBJECT_MODE** at the time of invocation. For example, you may not be aware that **OBJECT_MODE** has been set to **64**, and you may unexpectedly obtain 64-bit object files.

We strongly urge you to be aware of the setting of **OBJECT_MODE** at all times and to set **OBJECT_MODE** yourself to ensure that the compiler is invoked for the correct bit mode. Instead, you can always use the **-q32** or **-q64** option to specify the bit mode.

Module Support

64-bit support is provided in the Fortran module files that are shipped with XL Fortran. The 64-bit and 32-bit Fortran modules are shipped in the **/usr/lpp/xlf/include** directory.

XL Fortran Floating-Point Processing

This section answers some common questions about floating-point processing, such as:
- How can I get predictable, consistent results?
- How can I get the fastest or the most accurate results?
- How can I detect, and possibly recover from, exception conditions?
- Which compiler options can I use for floating-point calculations?

Related Information: This section makes frequent reference to the compiler options that are grouped together in "Options for Floating-Point Processing" on page 86, especially the "-qfloat Option" on page 163. The XL Fortran compiler also provides three intrinsic modules for exception handling and IEEE arithmetic support to help you write IEEE module-compliant code that can be more portable. See *IEEE Modules and Support* in the *XL Fortran Enterprise Edition for AIX Language Reference* for details.

The use of the compiler options for floating-point calculations affects the accuracy, performance, and possibly the correctness of floating-point calculations. Although the default values for the options were chosen to provide efficient and correct execution of most programs, you may need to specify nondefault options for your applications to work the way you want. We strongly advise you to read this section before using these options.

Note: The discussions of single-, double-, and extended-precision calculations in this section all refer to the default situation, with **-qrealsize=4** and no **-qautodbl** specified. If you change these settings, keep in mind that the size of a Fortran **REAL**, **DOUBLE PRECISION**, and so on may change, but single precision, double precision, and extended precision (in lowercase) still refer to 4-, 8-, and 16-byte entities respectively.

The information in this section relates to floating-point processing on the PowerPC family of processors. The section "Floating-Point Processing on the POWER and POWER2 Architectures" on page 303 describes the differences between floating-point processing on the PowerPC processors and floating-point processing on the POWER and POWER2 processors.

IEEE Floating-Point Overview

Here is a brief summary of the *IEEE Standard for Floating-Point Arithmetic* and the details of how it applies to XL Fortran on specific hardware platforms. For information on the draft Fortran 2003 IEEE Module and arithmetic support, see the *XL Fortran Enterprise Edition for AIX Language Reference*.

Compiling for Strict IEEE Conformance

By default, XL Fortran follows most, but not all of the rules in the IEEE standard. To compile for strict compliance with the standard:
- Use the compiler option **-qfloat=nomaf**.
- If the program changes the rounding mode at run time, include **rrm** among the **-qfloat** suboptions.

- If the data or program code contains signaling NaN values (NaNS), include **nans** among the **-qfloat** suboptions. (A signaling NaN is different from a quiet NaN; you must explicitly code it into the program or data or create it by using the **-qinitauto** compiler option.)
- If compiling with **-O3**, include the option **-qstrict** also.

IEEE Single- and Double-Precision Values

XL Fortran encodes single-precision and double-precision values in IEEE format. For the range and representation, see *Real* in the *XL Fortran Enterprise Edition for AIX Language Reference*.

IEEE Extended-Precision Values

The IEEE standard suggests, but does not mandate, a format for extended-precision values. XL Fortran does not use this format. “Extended-Precision Values” on page 291 describes the format that XL Fortran uses.

Infinities and NaNs

For single-precision real values:
- Positive infinity is represented by the bit pattern X'7F80 0000'.
- Negative infinity is represented by the bit pattern X'FF80 0000'.
- A signaling NaN is represented by any bit pattern between X'7F80 0001' and X'7FBF FFFF' or between X'FF80 0001' and X'FFBF FFFF'.
- A quiet NaN is represented by any bit pattern between X'7FC0 0000' and X'7FFF FFFF' or between X'FFC0 0000' and X'FFFF FFFF'.

For double-precision real values:
- Positive infinity is represented by the bit pattern X'7FF00000 00000000'.
- Negative infinity is represented by the bit pattern X'FFF00000 00000000'.
- A signaling NaN is represented by any bit pattern between X'7FF00000 00000001' and X'7FF7FFFF FFFFFFFF' or between X'FFF00000 00000001' and X'FFF7FFFF FFFFFFFF'.
- A quiet NaN is represented by any bit pattern between X'7FF80000 00000000' and X'7FFFFFFF FFFFFFFF' or between X'FFF80000 00000000' and X'FFFFFFFF FFFFFFFF'.

These values do not correspond to any Fortran real constants. You can generate all of these by encoding the bit pattern directly, or by using the **ieee_value** function provided in the **ieee_arithmetic** module. Using the **ieee_value** function is the preferred programming technique, as it is allowed by the Fortran 2003 draft standard and the results are portable. Encoding the bit pattern directly could cause portability problems on machines using different bit patterns for the different values. All except signaling NaN values can occur as the result of arithmetic operations:

```
$ cat fp_values.f
real plus_inf, minus_inf, plus_nanq, minus_nanq, nans
real large

data plus_inf /z'7f800000'/
data minus_inf /z'ff800000'/
data plus_nanq /z'7fc00000'/
data minus_nanq /z'ffc00000'/
data nans /z'7f800001'/

print *, 'Special values:', plus_inf, minus_inf, plus_nanq, minus_nanq, nans

! They can also occur as the result of operations.
large = 10.0 ** 200
print *, 'Number too big for a REAL:', large * large
print *, 'Number divided by zero:', (-large) / 0.0
print *, 'Nonsensical results:', plus_inf - plus_inf, sqrt(-large)

! To find if something is a NaN, compare it to itself.
print *, 'Does a quiet NaN equal itself:', plus_nanq .eq. plus_nanq
print *, 'Does a signaling NaN equal itself:', nans .eq. nans
! Only for a NaN is this comparison false.

end
$ xlf95 -o fp_values fp_values.f
** _main   === End of Compilation 1 ===
1501-510  Compilation successful for file fp_values.f.
$ fp_values
 Special values: INF -INF NaNQ -NaNQ NaNS
 Number too big for a REAL: INF
 Number divided by zero: -INF
 Nonsensical results: NaNQ NaNQ
 Does a quiet NaN equal itself: F
 Does a signaling NaN equal itself: F
```

Exception-Handling Model

The IEEE standard defines several exception conditions that can occur:

OVERFLOW
The exponent of a value is too large to be represented.

UNDERFLOW
A nonzero value is so small that it cannot be represented without an extraordinary loss of accuracy. The value can be represented only as zero or a denormal number.

ZERODIVIDE
A finite nonzero value is divided by zero.

INVALID
Operations are performed on values for which the results are not defined. These include:
- Operations on signaling NaN values
- infinity - infinity
- 0.0 * infinity
- 0.0 / 0.0
- mod(x,y) or ieee_rem(x,y) (or other remainder functions) when x is infinite or y is zero
- The square root of a negative number
- Conversion of a floating point number to an integer when the converted value cannot be represented faithfully

- Comparisons involving NaN values

INEXACT
A computed value cannot be represented exactly, so a rounding error is introduced. (This exception is very common.)

XL Fortran always detects these exceptions when they occur, but the default is not to take any special action. Calculation continues, usually with a NaN or infinity value as the result. If you want to be automatically informed when an exception occurs, you can turn on exception trapping through compiler options or calls to intrinsic subprograms. However, different results, intended to be manipulated by exception handlers, are produced:

Table 17. Results of IEEE Exceptions, with and without Trapping Enabled

	Overflow	**Underflow**	**Zerodivide**	**Invalid**	**Inexact**
Exceptions not enabled (default)	INF	Denormalized number	INF	NaN	Rounded result
Exceptions enabled	Unnormalized number with biased exponent	Unnormalized number with biased exponent	No result	No result	Rounded result

Note: Because different results are possible, it is very important to make sure that any exceptions that are generated are handled correctly. See "Detecting and Trapping Floating-Point Exceptions" on page 296 for instructions on doing so.

Hardware-Specific Floating-Point Overview

Single- and Double-Precision Values

The PowerPC floating-point hardware performs calculations in either IEEE single-precision (equivalent to **REAL(4)** in Fortran programs) or IEEE double-precision (equivalent to **REAL(8)** in Fortran programs).

Keep the following considerations in mind:

- Double precision provides greater range (approximately 10**(-308) to 10**308) and precision (about 15 decimal digits) than single precision (approximate range 10**(-38) to 10**38, with about 7 decimal digits of precision).
- Computations that mix single and double operands are performed in double precision, which requires conversion of the single-precision operands to double-precision. These conversions do not affect performance.
- Double-precision values that are converted to single-precision (such as when you specify the **SNGL** intrinsic or when a double-precision computation result is stored into a single-precision variable) require rounding operations. A rounding operation produces the correct single-precision value, which is based on the IEEE rounding mode in effect. The value may be less precise than the original double-precision value, as a result of rounding error. Conversions from double-precision values to single-precision values may reduce the performance of your code.
- Programs that manipulate large amounts of floating-point data may run faster if they use **REAL(4)** rather than **REAL(8)** variables. (You need to ensure that **REAL(4)** variables provide you with acceptable range and precision.) The programs may run faster because the smaller data size reduces memory traffic, which can be a performance bottleneck for some applications.

The floating-point hardware also provides a special set of double-precision operations that multiply two numbers and add a third number to the product. These combined multiply-add (**MAF**) operations are performed at the same speed that either a multiply or an add operation alone is performed. The **MAF** functions provide an extension to the IEEE standard because they perform the multiply and add with one (rather than two) rounding errors. The **MAF** functions are faster and more accurate than the equivalent separate operations.

Extended-Precision Values

XL Fortran extended precision is not in the format suggested by the IEEE standard, which suggests extended formats using more bits in both the exponent (for greater range) and the fraction (for greater precision).

XL Fortran extended precision, equivalent to **REAL(16)** in Fortran programs, is implemented in software. Extended precision provides the same range as double precision (about 10**(-308) to 10**308) but more precision (a variable amount, about 31 decimal digits or more). The software support is restricted to round-to-nearest mode. Programs that use extended precision must ensure that this rounding mode is in effect when extended-precision calculations are performed. See "Selecting the Rounding Mode" on page 292 for the different ways you can control the rounding mode.

Programs that specify extended-precision values as hexadecimal, octal, binary, or Hollerith constants must follow these conventions:

- Extended-precision numbers are composed of two double-precision numbers with different magnitudes that do not overlap. That is, the binary exponents differ by at least the number of fraction bits in a **REAL(8)**. The high-order double-precision value (the one that comes first in storage) must have the larger magnitude. The value of the extended-precision number is the sum of the two double-precision values.
- For a value of NaN or infinity, you must encode one of these values within the high-order double-precision value. The low-order value is not significant.

Because an XL Fortran extended-precision value can be the sum of two values with greatly different exponents, leaving a number of assumed zeros in the fraction, the format actually has a variable precision with a minimum of about 31 decimal digits. You get more precision in cases where the exponents of the two double values differ in magnitude by more than the number of digits in a double-precision value. This encoding allows an efficient implementation intended for applications requiring more precision but no more range than double precision.

Notes:

1. In the discussions of rounding errors because of compile-time folding of expressions, keep in mind that this folding produces different results for extended-precision values more often than for other precisions.
2. Special numbers, such as NaN and infinity, are not fully supported for extended-precision values. Arithmetic operations do not necessarily propagate these numbers in extended precision.
3. XL Fortran does not always detect floating-point exception conditions (see "Detecting and Trapping Floating-Point Exceptions" on page 296) for extended-precision values. If you turn on floating-point exception trapping in programs that use extended precision, XL Fortran may also generate signals in cases where an exception condition does not really occur.

How XL Fortran Rounds Floating-Point Calculations

Understanding rounding operations in XL Fortran can help you get predictable, consistent results. It can also help you make informed decisions when you have to make tradeoffs between speed and accuracy.

In general, floating-point results from XL Fortran programs are more accurate than those from other implementations because of **MAF** operations and the higher precision used for intermediate results. If identical results are more important to you than the extra precision and performance of the XL Fortran defaults, read "Duplicating the Floating-Point Results of Other Systems" on page 295.

Selecting the Rounding Mode

To change the rounding mode in a program, you can call the **fpsets** and **fpgets** routines, which use an array of logicals named **fpstat**, defined in the include files **/usr/include/fpdt.h** and **/usr/include/fpdc.h**. The **fpstat** array elements correspond to the bits in the floating-point status and control register.

For floating-point rounding control, the array elements **fpstat(fprn1)** and **fpstat(fprn2)** are set as specified in the following table:

Table 18. Rounding-Mode Bits to Use with fpsets and fpgets

fpstat(fprn1)	fpstat(fprn2)	Rounding Mode Enabled
.true.	.true.	Round towards -infinity.
.true.	.false.	Round towards +infinity.
.false.	.true.	Round towards zero.
.false.	.false.	Round to nearest.

For example:

```
      program fptest
        include 'fpdc.h'

        call fpgets( fpstat ) ! Get current register values.
        if ( (fpstat(fprn1) .eqv. .false.) .and. +
               (fpstat(fprn2) .eqv. .false.)) then
        print *, 'Before test: Rounding mode is towards nearest'
        print *, '             2.0 / 3.0 = ', 2.0 / 3.0
        print *, '            -2.0 / 3.0 = ', -2.0 / 3.0
        end if

        call fpgets( fpstat )   ! Get current register values.
        fpstat(fprn1) = .TRUE.  ! These 2 lines mean round towards
        fpstat(fprn2) = .FALSE. !   +INFINITY.
        call fpsets( fpstat )
        r = 2.0 / 3.0
        print *, 'Round towards +INFINITY:  2.0 / 3.0= ', r

        call fpgets( fpstat )   ! Get current register values.
        fpstat(fprn1) = .TRUE.  ! These 2 lines mean round towards
        fpstat(fprn2) = .TRUE.  !   -INFINITY.
        call fpsets( fpstat )
        r = -2.0 / 3.0
        print *, 'Round towards -INFINITY: -2.0 / 3.0= ', r
        end
! This block data program unit initializes the fpstat array, and so on.
```

```
  block data
  include 'fpdc.h'
  include 'fpdt.h'
end
```

XL Fortran also provides several procedures that allow you to control the floating-point status and control register of the processor directly. These procedures are more efficient than the **fpsets** and **fpgets** subroutines because they are mapped into inlined machine instructions that manipulate the floating-point status and control register (fpscr) directly.

XL Fortran supplies the **get_round_mode()** and **set_round_mode()** procedures in the **xlf_fp_util** module. These procedures return and set the current floating-point rounding mode, respectively.

For example:

```
program fptest
  use, intrinsic :: xlf_fp_util
  integer(fpscr_kind) old_fpscr
  if ( get_round_mode() == fp_rnd_rn ) then
  print *, 'Before test: Rounding mode is towards nearest'
  print *, '             2.0 / 3.0 = ', 2.0 / 3.0
  print *, '            -2.0 / 3.0 = ', -2.0 / 3.0
  end if

  old_fpscr = set_round_mode( fp_rnd_rp )
  r = 2.0 / 3.0
  print *, 'Round towards +infinity:  2.0 / 3.0 = ', r

  old_fpscr = set_round_mode( fp_rnd_rm )
  r = -2.0 / 3.0
  print *, 'Round towards -infinity: -2.0 / 3.0 = ', r
end
```

XL Fortran supplies the **ieee_get_rounding_mode()** and **ieee_set_rounding_mode()** procedures in the **ieee_arithmetic** module. These portable procedures retrieve and set the current floating-point rounding mode, respectively.

For example:

```
program fptest
  use, intrinsic :: ieee_arithmetic
  type(ieee_round_type) current_mode
  call ieee_get_rounding_mode( current_mode )
  if ( current_mode == ieee_nearest ) then
  print *, 'Before test: Rounding mode is towards nearest'
  print *, '             2.0 / 3.0 = ', 2.0 / 3.0
  print *, '            -2.0 / 3.0 = ', -2.0 / 3.0
  end if

  call ieee_set_rounding_mode( ieee_up )
  r = 2.0 / 3.0
  print *, 'Round towards +infinity:  2.0 / 3.0 = ', r

  call ieee_set_rounding_mode( ieee_down )
  r = -2.0 / 3.0
  print *, 'Round towards -infinity: -2.0 / 3.0 = ', r
end
```

Notes:

1. Extended-precision floating-point values must only be used in round-to-nearest mode.

2. For thread-safety and reentrancy, the include file **/usr/include/fpdc.h** contains a **THREADLOCAL** directive that is protected by the trigger constant **IBMT**. The invocation commands **xlf_r**, **xlf_r7**, **xlf90_r**, **xlf90_r7**, **xlf95_r**, and **xlf95_r7** turn on the **-qthreaded** compiler option by default, which in turn implies the trigger constant **IBMT**. If you are including the file **/usr/include/fpdc.h** in code that is not intended to be thread-safe, do not specify **IBMT** as a trigger constant.

Related Information: For more information about the bits in the FPSCR register that correspond to the **fpstat** array elements, see the *POWERstation and POWERserver® Hardware Technical Reference - General Information*.

Minimizing Rounding Errors

There are several strategies for handling rounding errors and other unexpected, slight differences in calculated results. You may want to consider one or more of the following strategies:
- Minimizing the amount of overall rounding
- Delaying as much rounding as possible to run time
- Ensuring that if some rounding is performed in a mode other than round-to-nearest, *all* rounding is performed in the same mode

Minimizing Overall Rounding

Rounding operations, especially in loops, reduce code performance and may have a negative effect on the precision of computations. Consider using double-precision variables instead of single-precision variables when you store the temporary results of double-precision calculations, and delay rounding operations until the final result is computed. You can also specify the **hssngl** suboption of **-qfloat** instead of converting a stored single-precision result back to double-precision. This suboption preserves computed double-precision results so that they can be used again later.

Delaying Rounding until Run Time

The compiler evaluates floating-point expressions during compilation when it can, so that the resulting program does not run more slowly due to unnecessary run-time calculations. However, the results of the compiler's evaluation might not match exactly the results of the run-time calculation. To delay these calculations until run time, specify the **nofold** suboption of the **-qfloat** option.

The results may still not be identical; for example, calculations in **DATA** and **PARAMETER** statements are still performed at compile time.

The differences in results due to **fold** or **nofold** are greatest for programs that perform extended-precision calculations or are compiled with the **-O** option or both.

Ensuring that the Rounding Mode is Consistent

You can change the rounding mode from its default setting of round-to-nearest. (See for examples.) If you do so, you must be careful that *all* rounding operations for the program use the same mode:
- Specify the equivalent setting on the **-qieee** option, so that any compile-time calculations use the same rounding mode.
- Specify the **rrm** suboption of the **-qfloat** option, so that the compiler does not perform any optimizations that require round-to-nearest rounding mode to work correctly.

For example, you might compile a program like the one in "Selecting the Rounding Mode" on page 292 with this command if the program consistently uses round-to-plus-infinity mode:

```
xlf95 -qieee=plus -qfloat=rrm changes_rounding_mode.f
```

Duplicating the Floating-Point Results of Other Systems

To duplicate the double-precision results of programs on systems with different floating-point architectures (without multiply-add instructions), specify the **nomaf** suboption of the **-qfloat** option. This suboption prevents the compiler from generating any multiply-add operations. This results in decreased accuracy and performance but provides strict conformance to the IEEE standard for double-precision arithmetic.

To duplicate the results of programs where the default size of **REAL** items is different from that on systems running XL Fortran, use the **-qrealsize** option (page 222) to change the default **REAL** size when compiling with XL Fortran.

If the system whose results you want to duplicate preserves full double precision for default real constants that are assigned to **DOUBLE PRECISION** variables, use the **-qdpc** or **-qrealsize** option.

If results consistent with other systems are important to you, include **norsqrt** and **nofold** in the settings for the **-qfloat** option. If you specify the option **-O3**, include **-qstrict** too.

Maximizing Floating-Point Performance

If performance is your primary concern and you want your program to be relatively safe but do not mind if results are slightly different (generally more precise) from what they would be otherwise, optimize the program with the **-O** option, and specify **-qfloat=rsqrt:hssngl:fltint**. The following section describes the functions of these suboptions:

- The **rsqrt** suboption replaces division by a square root with multiplication by the reciprocal of the root, a faster operation that may not produce precisely the same result.
- The **hssngl** suboption is the opposite of **rndsngl**; it improves the performance of single-precision (**REAL(4)**) floating-point calculations by suppressing rounding operations that are required by the Fortran language but are not necessary for correct program execution. The results of floating-point expressions are kept in double precision where the original program would round them to single-precision. These results are then used in later expressions instead of the rounded results.

  To detect single-precision floating-point overflows and underflows, rounding operations are still inserted when double-precision results are stored into single-precision memory locations. However, if optimization removes such a store operation, **hssngl** also removes the corresponding rounding operation, possibly preventing the exception. (Depending on the characteristics of your program, you may or may not care whether the exception happens.)

  The **hssngl** suboption is safe for all types of programs because it always only *increases* the precision of floating-point calculations. Program results may differ because of the increased precision and because of avoidance of some exceptions.

- The **fltint** suboption speeds up float-to-integer conversions by reducing error checking for overflows. You should make sure that any floats that are converted to integers are not outside the range of the corresponding integer types.

In cases where speed is so important that you can make an informed decision to sacrifice correctness at boundary conditions, you can replace **hssngl** and **fltint** with the **hsflt** suboption; it does the same thing as **fltint** and suppresses rounding operations.

In suppressing rounding operations, **hsflt** works like **hssngl**, but it also suppresses rounding operations when double-precision values are assigned to single-precision memory locations. Single-precision overflow is not detected in such assignments, and the assigned value is not correctly rounded according to the current rounding mode.

Attention: When you use the **hsflt** suboption, observe these restrictions, or your program may produce incorrect results without warning:
- Your program must never convert overly large floats to integer.
- Your program must never compute NaNs, or values outside the range of single precision.
- Your program must not depend on results to be correctly rounded to single precision: for example, by comparing two single-precision values for equality.

Therefore, we recommend that you use this suboption only with extreme caution. It is for use by knowledgeable programmers in specific applications, such as graphics programs, where the computational characteristics are known. If you are at all unsure whether a program is suitable or if the program produces unexpected results when you use this suboption, use **hssngl** instead.

Related Information: "Technical Details of the -qfloat=hsflt Option" on page 412 provides additional technical information about this suboption.

Detecting and Trapping Floating-Point Exceptions

As stated earlier, the IEEE standard for floating-point arithmetic defines a number of exception (or error) conditions that might require special care to avoid or recover from. The following sections are intended to help you make your programs work safely in the presence of such exception conditions while sacrificing the minimum amount of performance.

The floating-point hardware always detects a number of floating-point exception conditions (which the IEEE standard rigorously defines): overflow, underflow, zerodivide, invalid, and inexact.

By default, the only action that occurs is that a status flag is set. The program continues without a problem (although the results from that point on may not be what you expect). If you want to know when an exception occurs, you can arrange for one or more of these exception conditions to generate a signal.

The signal causes a branch to a handler routine. The handler receives information about the type of signal and the state of the program when the signal occurred. It can produce a core dump, display a listing showing where the exception occurred, modify the results of the calculation, or carry out some other processing that you specify.

The XL Fortran compiler and the operating system provide facilities for working with floating-point exception conditions. The compiler facilities indicate the presence of exceptions by generating **SIGTRAP** signals. The operating-system facilities generate **SIGFPE** signals. Do not mix these different facilities within a single program.

Compiler Features for Trapping Floating-Point Exceptions

To turn on XL Fortran exception trapping, compile the program with the **-qflttrap** option and some combination of suboptions that includes **enable**. This option uses trap operations to detect floating-point exceptions and generates **SIGTRAP** signals when exceptions occur.

-qflttrap also has suboptions that correspond to the names of the exception conditions. For example, if you are only concerned with handling overflow and underflow exceptions, you could specify something similar to the following:

```
xlf95 -qflttrap=overflow:underflow:enable compute_pi.f
```

You only need **enable** when you are compiling the main program. However, it is very important and does not cause any problems if you specify it for other files, so always include it when you use **-qflttrap**.

An advantage of this approach is that performance impact is relatively low. To further reduce performance impact, you can include the **imprecise** suboption of the **-qflttrap** option. This suboption delays any trapping until the program reaches the start or end of a subprogram.

The disadvantages of this approach include the following:

- It only traps exceptions that occur in code that you compiled with **-qflttrap**, which does not include system library routines.
- It is generally not possible for a handler to substitute results for failed calculations if you use the **imprecise** suboption of **-qflttrap**.

Notes:

1. If your program depends on floating-point exceptions occurring for particular operations, also specify **-qfloat** suboptions that include **nofold** and **nohssngl**. Otherwise, the compiler might replace an exception-producing calculation with a constant NaN or infinity value, or it might eliminate an overflow in a single-precision operation.
2. The suboptions of the **-qflttrap** option replace an earlier technique that required you to modify your code with calls to the **fpsets** and **fpgets** procedures. You no longer require these calls for exception handling if you use the appropriate **-qflttrap** settings.

   Attention: If your code contains **fpsets** calls that enable checking for floating-point exceptions and you do not use the **-qflttrap** option when compiling the whole program, the program will produce unexpected results if exceptions occur, as explained in Table 17 on page 290.

Operating System Features for Trapping Floating-Point Exceptions

A direct way to turn on exception trapping is to call the operating system routine **fp_trap**. It uses the system hardware to detect floating-point exceptions and generates **SIGFPE** signals when exceptions occur. Fortran definitions for the values needed to call it are in the files **/usr/include/fp_fort_c.f**, **fp_fort_t.f**, or the **xlf_fp_util** module.

There are other related operating system routines that you can locate by reading the description of **fp_trap**.

The advantages of this approach include:
- It works for any code, regardless of the language and without the need to compile with any special options.
- It generates **SIGFPE** signals, the same as other popular Unix systems.

The disadvantages of this approach include:
- The program may run much slower while exception checking is turned on.
- The call to **FP_TRAP** requires a source-code change and thus a recompilation.

Installing an Exception Handler

When a program that uses the XL Fortran or AIX exception-detection facilities encounters an exception condition, it generates a signal. This causes a branch to whatever handler is specified by the program. The information in this section, except the explanation of the **-qsigtrap** option, applies both to **SIGTRAP** and **SIGFPE** signals.

By default, the program stops after producing a core file, which you can use with a debugger to locate the problem. If you want to install a **SIGTRAP** signal handler, use the **-qsigtrap** option. It allows you to specify an XL Fortran handler that produces a traceback or to specify a handler you have written:

```
xlf95 -qflttrap=ov:und:en pi.f                               # Dump core on an exception
xlf95 -qflttrap=ov:und:en -qsigtrap pi.f                     # Uses the xl__trce handler
xlf95 -qflttrap=ov:und:en -qsigtrap=return_22_over_7 pi.f  # Uses any other handler
```

You can also install an alternative exception handler, either one supplied by XL Fortran or one you have written yourself, by calling the **SIGNAL** subroutine (defined in **/usr/include/fexcp.h**):

```
  INCLUDE 'fexcp.h'
  CALL SIGNAL(SIGTRAP,handler_name)
  CALL SIGNAL(SIGFPE,handler_name)
```

The XL Fortran exception handlers and related routines are:

xl__ieee
: Produces a traceback and an explanation of the signal and continues execution by supplying the default IEEE result for the failed computation. This handler allows the program to produce the same results as if exception detection was not turned on.

xl__trce
: Produces a traceback and stops the program.

xl__trcedump
: Produces a traceback and a core file and stops the program.

xl__sigdump
: Provides a traceback that starts from the point at which it is called and provides information about the signal. You can only call it from inside a user-written signal handler, and it requires the same parameters as other AIX signal handlers. It does not stop the program. To successfully continue, the signal handler must perform some cleanup after calling this subprogram.

xl__trbk
: Provides a traceback that starts from the point at which it is called. You call it as a subroutine from

your code, rather than specifying it with the **-qsigtrap** option. It requires no parameters. It does not stop the program.

All of these handler names contain double underscores to avoid duplicating names that you declared in your program. All of these routines work for both **SIGTRAP** and **SIGFPE** signals.

You can use the **-g** compiler option to get line numbers in the traceback listings. The file **/usr/include/fsignal.h** defines a Fortran derived type similar to the sigcontext structure in **/usr/include/sys/signal.h**. You can write a Fortran signal handler that accesses this derived type.

Related Information: "Sample Programs for Exception Handling" on page 302 lists some sample programs that illustrate how to use these signal handlers or write your own. For more information, see **SIGNAL**, in the *Intrinsic Procedures* section in the *XL Fortran Enterprise Edition for AIX Language Reference*.

Producing a Core File

To produce a core file, do not install an exception handler, or else specify the **xl__trcedump** handler.

Controlling the Floating-Point Status and Control Register

Before the **-qflttrap** suboptions or the **-qsigtrap** options, most of the processing for floating-point exceptions required you to change your source files to turn on exception trapping or install a signal handler. Although you can still do so, for any new applications, we recommend that you use the options instead.

To control exception handling at run time, compile without the **enable** suboption of the **-qflttrap** option:

```
xlf95 -qflttrap compute_pi.f     # Check all exceptions, but do not trap.
xlf95 -qflttrap=ov compute_pi.f  # Check one type, but do not trap.
```

Then, inside your program, manipulate the **fpstats** array (defined in the include file **/usr/include/fpdc.h**) and call the **fpsets** subroutine to specify which exceptions should generate traps.

See the sample program that uses **fpsets** and **fpgets** in "Selecting the Rounding Mode" on page 292.

Another method is to use the **set_fpscr_flags()** subroutine in the **xlf_fp_util** module. This subroutine allows you to set the floating-point status and control register flags you specify in the **MASK** argument. Flags that you do not specify in **MASK** remain unaffected. MASK must be of type **INTEGER(FPSCR_KIND)**. For example:

```
USE, INTRINSIC :: xlf_fp_util
INTEGER(FPSCR_KIND) SAVED_FPSCR
INTEGER(FP_MODE_KIND) FP_MODE

SAVED_FPSCR = get_fpscr()           ! Saves the current value of
                                    ! the fpscr register.

CALL set_fpscr_flags(TRP_DIV_BY_ZERO) ! Enables trapping of
! ...                                 ! divide-by-zero.
SAVED_FPSCR=set_fpscr(SAVED_FPSCR)    ! Restores fpscr register.
```

Another method is to use the **ieee_set_halting_mode** subroutine in the **ieee_exceptions** module. This portable, elemental subroutine allows you to set the halting (trapping) status for any **FPSCR** exception flags. For example:

```
USE, INTRINSIC :: ieee_exceptions
 TYPE(IEEE_STATUS_TYPE) SAVED_FPSCR
 CALL ieee_get_status(SAVED_FPSCR)   ! Saves the current value of the
                                     ! fpscr register

 CALL ieee_set_halting_mode(IEEE_DIVIDE_BY_ZERO, .TRUE.)  ! Enabled trapping
 ! ...                                                    ! of divide-by-zero.

 CALL IEEE_SET_STATUS(SAVED_FPSCR)  ! Restore fpscr register
```

xlf_fp_util Procedures

The **xlf_fp_util** procedures allow you to query and control the floating-point status and control register (fpscr) of the processor directly. These procedures are more efficient than the **fpsets** and **fpgets** subroutines because they are mapped into inlined machine instructions that manipulate the floating-point status and control register directly.

The intrinsic module, **xlf_fp_util**, contains the interfaces and data type definitions for these procedures and the definitions for the named constants that are needed by the procedures. This module enables type checking of these procedures at compile time rather than link time. The following files are supplied for the **xlf_fp_util** module:

File names	File type	Locations
xlf_fp_util.mod	module symbol file (32–bit)	/usr/lpp/xlf/include_d7 /usr/lpp/xlf/include
	module symbol file (64–bit)	/usr/lpp/xlf/include

To use the procedures, you must add a **USE XLF_FP_UTIL** statement to your source file. For more information, see **USE**, in the *XL Fortran Enterprise Edition for AIX Language Reference*.

When compiling with the **-U** option, you must code the names of these procedures in all lowercase.

For a list of the **xlf_fp_util** procedures, see the *Service and Utility Procedures* section in the *XL Fortran Enterprise Edition for AIX Language Reference*.

fpgets and fpsets Subroutines

The **fpsets** and **fpgets** subroutines provide a way to manipulate or query the floating-point status and control register. Instead of calling the operating system routines directly, you pass information back and forth in **fpstat**, an array of logicals. The following table shows the most commonly used array elements that deal with exceptions:

Table 19. Exception Bits to Use with fpsets and fpgets

Array Element to Set to Enable	Array Element to Check if Exception Occurred	Exception Indicated When .TRUE.
n/a	fpstat(fpfx)	Floating-point exception summary

Table 19. Exception Bits to Use with fpsets and fpgets (continued)

Array Element to Set to Enable	Array Element to Check if Exception Occurred	Exception Indicated When .TRUE.
n/a	fpstat(fpfex)	Floating-point enabled exception summary
fpstat(fpve)	fpstat(fpvx)	Floating-point invalid operation exception summary
fpstat(fpoe)	fpstat(fpox)	Floating-point overflow exception
fpstat(fpue)	fpstat(fpux)	Floating-point underflow exception
fpstat(fpze)	fpstat(fpzx)	Zero-divide exception
fpstat(fpxe)	fpstat(fpxx)	Inexact exception
fpstat(fpve)	fpstat(fpvxsnan)	Floating-point invalid operation exception (NaNS)
fpstat(fpve)	fpstat(fpvxisi)	Floating-point invalid operation exception (INF-INF)
fpstat(fpve)	fpstat(fpvxidi)	Floating-point invalid operation exception (INF/INF)
fpstat(fpve)	fpstat(fpvxzdz)	Floating-point invalid operation exception (0/0)
fpstat(fpve)	fpstat(fpvximz)	Floating-point invalid operation exception (INF*0)
fpstat(fpve)	fpstat(fpvxvc)	Floating-point invalid operation exception (invalid compare)
n/a	fpstat(fpvxsoft)	Floating-point invalid operation exception (software request), PowerPC only
n/a	fpstat(fpvxsqrt)	Floating-point invalid operation exception (invalid square root), PowerPC only
n/a	fpstat(fpvxcvi)	Floating-point invalid operation exception (invalid integer convert), PowerPC only

To explicitly check for specific exceptions at particular points in a program, use **fpgets** and then test whether the elements in **fpstat** have changed. Once an exception has occurred, the corresponding exception bit (second column in the preceding table) is set until it is explicitly reset, except for **fpstat(fpfx)**, **fpstat(fpvx)**, and **fpstat(fpfex)**, which are reset only when the specific exception bits are reset.

An advantage of using the **fpgets** and **fpsets** subroutines (as opposed to controlling everything with suboptions of the **-qflttrap** option) includes control over granularity of exception checking. For example, you might only want to test if an exception occurred anywhere in the program when the program ends.

The disadvantages of this approach include the following:
- You have to change your source code.
- These routines differ from what you may be accustomed to on other platforms.

For example, to trap floating-point overflow exceptions but only in a certain section of the program, you would set `fpstat(fpoe)` to `.TRUE.` and call **fpsets**. After the exception occurs, the corresponding exception bit, `fpstat(fpox)`, is `.TRUE.` until the program runs:

```
call fpgets(fpstat)
fpstat(fpox) = .FALSE.
call fpsets(fpstat)   ! resetting fpstat(fpox) to .FALSE.
```

Sample Programs for Exception Handling

/usr/lpp/xlf/samples/floating_point contains a number of sample programs to illustrate different aspects of exception handling:

flttrap_handler.c and **flttrap_test.f**
: A sample exception handler that is written in C and a Fortran program that uses it.

xl__ieee.F and **xl__ieee.c**
: Exception handlers that are written in Fortran and C that show how to substitute particular values for operations that produce exceptions. Even when you use support code such as this, the implementation of XL Fortran exception handling does not fully support the exception-handling environment that is suggested by the IEEE floating-point standard.

check_fpscr.f and **postmortem.f**
: Show how to work with the **fpsets** and **fpgets** procedures and the **fpstats** array.

fhandler.F
: Shows a sample Fortran signal handler and demonstrates the **xl__sigdump** procedure.

xl__trbk_test.f
: Shows how to use the **xl__trbk** procedure to generate a traceback listing without stopping the program.

The sample programs are strictly for illustrative purposes only.

Causing Exceptions for Particular Variables

To mark a variable as "do not use", you can encode a special value called a signaling NaN in it. This causes an invalid exception condition any time that variable is used in a calculation.

If you use this technique, use the **nans** suboption of the **-qfloat** option, so that the program properly detects all cases where a signaling NaN is used, and one of the methods already described to generate corresponding **SIGFPE** or **SIGTRAP** signals.

Notes:
1. Because a signaling NaN is never generated as the result of a calculation and must be explicitly introduced to your program as a constant or in input data, you should not need to use this technique unless you deliberately use signaling NaN values in it.
2. In previous XL Fortran releases, the **-qfloat** suboption was called **spnans**. In the future, use **nans** instead (although **spnans** still works, for backward compatibility).

Minimizing the Performance Impact of Floating-Point Exception Trapping

If you need to deal with floating-point exception conditions but are concerned that doing so will make your program too slow, here are some techniques that can help minimize the performance impact:

- Consider using only a subset of the **overflow**, **underflow**, **zerodivide**, **invalid**, and **inexact** suboptions with the **-qflttrap** option if you can identify some conditions that will never happen or you do not care about. In particular, because an **inexact** exception occurs for each rounding error, you probably should not check for it if performance is important.
- Include the **imprecise** suboption with the **-qflttrap** option, so that your compiler command looks similar to this:

  ```
  xlf90 -qflttrap=underflow:enable:imprecise does_underflows.f
  ```

  imprecise makes the program check for the specified exceptions only on entry and exit to subprograms that perform floating-point calculations. This means that XL Fortran will eventually detect any exception, but you will know only the general area where it occurred, not the exact location.

  When you specify **-qflttrap** without **imprecise**, a check for exceptions follows each floating-point operation. If all your exceptions occur during calls to routines that are not compiled with **-qflttrap** (such as library routines), using **imprecise** is generally a good idea, because identifying the exact location will be difficult anyway.

  Note that **enable** has no effect if using the **nanq** suboption. **nanq** generates trapping code after each floating point arithmetic, load instruction and procedure returning floating point values even if **imprecise** is specified.

Floating-Point Processing on the POWER and POWER2 Architectures

The following section provides information on floating-point processing on the POWER and POWER2 processors.

Precision of Computations

POWER and POWER2 floating-point hardware performs all calculations in IEEE double-precision mode. The hardware does not directly perform single-precision calculations, but it is capable of generating single-precision results by using the following sequence of operations:

1. Convert all single-precision operands of a single-precision operation to double-precision.
2. Perform the equivalent double-precision operation.
3. Round the result to single-precision.

This sequence always produces exactly the same bit-for-bit result, as if the single-precision IEEE operation had been performed.

As on the PowerPC machines, conversions from single-precision to double-precision have no negative performance impacts, but rounding operations from double-precision to single-precision do. Since the performance penalty of rounding operations would normally impact all single-precision computations on a POWER or POWER2 machine, the compiler attempts to reduce the number of rounding operations. It does this under the control of the **norndsngl** suboption of the **-qfloat** option.

When you specify the **norndsgnl** suboption, the compiler leaves all intermediate results of single-precision operations in double-precision. That is, it suppresses the rounding operation in the above sequence. The compiler only performs a rounding operation on the final result of an expression, when it stores that result into a single-precision memory location.

The following example shows the difference between using the **norndsngl** and the **rndsngl** suboptions:

```
REAL(4) a,b,c,d
...
a = b + c + d
```

With **norndsngl**, the compiler does the following:

1. Performs the intermediate computation of `b + c` in double precision without rounding.
2. Adds the double-precision result to `d`.
3. Rounds the final double-precision result and then stores it into variable `a`.

With **rndsngl**, the compiler follows the same steps except that it performs rounding in the first step. Note that **norndsngl** maintains increased precision for intermediate results and improves performance, but you may need to specify **rndsngl** to produce results that are bit-for-bit identical to those computed on other systems.

norndsngl is the default when you use **-qarch** to target a POWER, a POWER2, or a common architecture. **rndsngl** is the default when you target a PowerPC architecture. You can also explicitly set the **rndsngl** suboption for any target architecture.

Invalid Operation Exceptions for SQRT Operations on POWER Processors

The POWER architecture does not include a hardware status flag to indicate IEEE invalid operation exceptions that are caused by attempting to compute the square root of a negative number. Instead, the operating system must handle those exceptions by using a software mechanism. Therefore, using **SQRT** for a negative number may not reliably generate invalid operation exceptions on POWER platforms, depending on the level of the operating system installed on the computer.

The POWER2 architecture and many PowerPC architectures each has a hardware status flag for invalid **SQRT** operations, and each reliably generates exceptions.

Optimizing XL Fortran Programs

This section provides background information on optimization, guidance on using XL Fortran's optimization features, and details of some XL Fortran optimization techniques.

Simple compilation is the translation or transformation of the source code into an executable or shared object. An optimizing transformation is one that gives your application better overall performance at run time. XL Fortran provides a portfolio of optimizing transformations tailored to the IBM hardware. These transformations can:

- Reduce the number of instructions executed for critical operations.
- Restructure the generated object code to make optimal use of the PowerPC architecture.
- Improve the usage of the memory subsystem.
- Exploit the ability of the architecture to handle large amounts of shared memory parallelization.

Significant performance improvements are possible with relatively little development effort because the compiler is capable of widely applicable and sophisticated program analysis and transformation. Moreover, the compilers enable programming models such as OpenMP, which allow you to write high-performance code.

Optimizations are intended for later phases of application development cycles, such as product release builds. If possible, you should test and debug your code without optimization before attempting to optimize it.

Optimization is controlled by compiler options and directives. However, compiler-friendly programming idioms can be as useful to performance as any of the options or directives. It is no longer necessary nor is it recommended to excessively hand-optimize your code (for example, manually unrolling loops). Unusual constructs can confuse the compiler (and other programmers), and make your application difficult to optimize for new machines. The section ***Compiler-Friendly Programming*** contains some suggested idioms and programming tips for writing good optimizable code.

It should be noted that not all optimizations are beneficial for all applications. A trade-off usually has to be made between an increase in compile time accompanied by reduced debugging capability and the degree of optimization done by the compiler.

The Philosophy of XL Fortran Optimizations

XL Fortran optimizations can be characterized according to their *aggressiveness*, which determines how much risk they carry. Only the very highest optimization levels perform aggressive optimizations, and even then the risk is limited to slightly different results in a small subset of possible programs.

The less-aggressive optimizations are intended to produce exactly the same results as an equivalent unoptimized program:

- Code that might cause an exception is not moved unless the exception is certain to occur anyway. In the following example, the program could evaluate the expression N/K before the loop because the result is the same for each iteration of the loop:

```
      DO 10 J=1,N
        ...
        IF (K .NE. 0) M(J)=N/K
        ...
10    END
```

  However, it is not moved because K might be 0, and computing **N/K** results in an exception where none occurs in the unoptimized program.
- The rules for IEEE arithmetic are followed more closely than otherwise. [3] For example, **X+0.0** is not folded to **X**, because IEEE rules require that **-0.0+0.0** be **0**, making **X+0** equal to **-X** in this one case.
- Floating-point calculations are not considered associative. For example, XL Fortran evaluates **X*Y*Z** left-to-right, even though the program might already have computed **Y*Z**, because the results might not be identical.

As the optimization level increases, these restrictions are relaxed where there is an opportunity for a performance improvement:

- Calculations like **N/K** in the previous example and floating-point operations may be moved or rescheduled because they are unlikely to cause exceptions.
- IEEE conformance is not enforced for rules that are unlikely to be needed. The sign of zero might not be correctly preserved, as in the preceding example. However, this might only be a problem in an extreme case, such as multiplying the wrongly signed zero by infinity and ending up with an infinity of the wrong sign. Floating-point operations that might cause an exception may be moved, rescheduled, or processed so they do not produce an exception.
- Floating-point expressions might be reassociated, so that results might not be identical.

When you specify the highest levels of optimization, XL Fortran assumes that you are requesting speed even at the possibility of some risk, as already explained. If you want as much optimization as possible without the resulting risk, you must add the **-qstrict** compiler option.

The early XL family of compilers adopted a conservative approach to optimization. This was intended to make an optimized program work exactly the same as an unoptimized one, even in extreme cases unlikely to occur in real life. For example, the array reference **A(N)** might not be optimized, because **N** might be a huge number so that the program causes a segmentation violation when the address is referenced, and this behavior would be "preserved". With the industry in general favoring a less conservative approach, XL Fortran's highest optimization levels now emphasize performance over identical execution between optimized and unoptimized programs.

The different levels of the **-O** option incorporate various optimization techniques that are expected to improve performance for many different kinds of programs. The specialized optimization options, such as **-qipa**, **-qhot**, and **-Q**, can improve performance in some kinds of programs but degrade it in others. Therefore, they may require experimentation to determine whether they are appropriate for any given program.

3. If IEEE compliance is a concern for you, you should also specify either the **-qfloat** or **-qstrict** option.

Summary of Compiler Options for Optimization

The following table describes the compiler options that have the most effect on performance. You can experiment with different combinations of compiler options to see which options are most suitable for your application.

Table 20. Compiler options for optimization

Option	Overview
-O0 or **-qnoopt**	The compiler performs very limited optimization. This is the default optimization level. Before you start optimizing your application, ensure that it compiles and executes successfully at optimization level 0.
-qarch	The compiler generates instructions for the specified processor architecture. This option allows the compiler to take advantage of processor instructions that exist on the specified architecture but do not exist on other architectures.
-qtune	The compiler optimizes the application for the specified processor architecture. The application will run on all of the processors specified by the **-qarch** option, but its performance will be tuned for the processors specified by the **-qtune** option.
-O2	The compiler performs basic optimization. This option provides a balance between compilation speed and run-time performance.
-O3	The compiler performs aggressive optimization, including optimizations that are memory-intensive, compile-time-intensive, or both.
-qhot	The compiler performs additional loop optimization, automatic vectorization, and optionally performs array padding. This option is most useful for scientific applications that contain numerical processing. If you want better performance than -O3 with less compile time than -O4, try **-O3 -qhot**.
-qcache	The compiler assumes that the processor will have the specified cache configuration. This option can improve performance if all of the processors that execute the application have the same non-default cache configuration.
-qipa	The compiler performs interprocedural analysis to optimize the entire application as a unit. This option is most useful for applications that contain a large number of frequently used routines. In many cases, this option significantly increases compilation time.
-O4	This is equivalent to: **-O3 -qipa -qhot -qarch=auto -qtune=auto -qcache=auto**
-O5	This is equivalent to: **-O4 -qipa=level=2**
-qpdf1 **-qpdf2**	The compiler uses profile-directed feedback to optimize the application based on an analysis of how often your application executes different sections of code.

Choosing an Optimization Level

Optimization requires additional compilation time, but usually results in a faster run time. XL Fortran allows you to select whether you want optimization to be performed at compile time. By default, the compiler performs very limited optimization (**-O0** or **-qnoopt**).

To enable compiler optimization, specify the **-O** compiler option with an optional digit that signifies the level. The following table summarizes compiler behavior at each optimization level.

Optimization levels

Option	Behavior
-qnoopt/-O0	Fast compilation, debuggable code, conserved program semantics.
-O2	Comprehensive low-level optimization; partial debugging support. (This is the same as specifying **-O**.)
-O3	More extensive optimization; some precision trade-offs.
-O4 and -O5	Interprocedural optimization; loop optimization; automatic machine tuning.

Optimization Level -O2

At optimization level **-O2** (same as **-O**), the compiler performs comprehensive low-level optimization, which includes the following techniques:.

- Global assignment of user variables to registers, also known as *graph coloring register allocation*.
- Strength reduction and effective use of addressing modes.
- Elimination of redundant instructions, also known as *common subexpression elimination*
- Elimination of instructions whose results are unused or that cannot be reached by a specified control flow, also known as *dead code elimination*.
- Value numbering (algebraic simplification).
- Movement of invariant code out of loops.
- Compile-time evaluation of constant expressions, also known as *constant propagation*.
- Control flow simplification.
- Instruction scheduling (reordering) for the target machine.
- Loop unrolling and software pipelining.

Minimal debugging information at optimization level **-O2** consists of the following behaviors:

- Externals and parameter registers are visible at procedure boundaries, which are the entrance and exit to a procedure. You can look at them if you set a breakpoint at the entry to a procedure. However, function inlining with **-Q** can eliminate these boundaries and this visibility. This can also happen when the compiler inlines very small functions.
- The **SNAPSHOT** directive creates additional program points for storage visibility by flushing registers to memory. This allows you to view and modify the values of any local or global variable, or of any parameter in your program. You can set a breakpoint at the **SNAPSHOT** and look at that particular area of storage in a debugger.
- The **-qkeepparm** option forces parameters to memory on entry to a procedure so that they can be visible in a stack trace.

Optimization Level -O3

At optimization level **-O3**, the compiler performs more extensive optimization than at **-O2**. The optimizations may be broadened or deepened in the following ways:

- Deeper inner loop unrolling.
- Better loop scheduling.
- Increased optimization scope, typically to encompass a whole procedure.
- Specialized optimizations (those that might not help all programs).
- Optimizations that require large amounts of compile time or space.
- Implicit memory usage limits are eliminated (equivalent to compiling with **-qmaxmem=-1**).

- Implies **-qnostrict**, which allows some reordering of floating-point computations and potential exceptions.

Due to the implicit setting of **-qnostrict**, some precision trade-offs are made by the compiler, such as the following:
- Reordering of floating-point computations.
- Reordering or elimination of possible exceptions (for example, division by zero, overflow).

-O3 optimizations may:
- Require more machine resources during compilation
- Take longer to compile
- Change the semantics of the program slightly

Use the **-O3** option where run-time performance is a crucial factor and machine resources can accommodate the extra compile-time work.

The exact optimizations that are performed depend on a number of factors:
- Whether the program can be rearranged and still execute correctly
- The relative benefit of each optimization
- The machine architecture

Getting the Most out of -O2 and -O3

Here is a recommended approach to using optimization levels **-O2** and **-O3**
- If possible, test and debug your code without optimization before using **-O2**.
- Ensure that your code complies with its language standard. Optimizers assume and rely on that fact that code is standard conformant. Code that is even subtly non-conformant can cause an optimizer to perform incorrect code transformations.

  Ensure that subroutine parameters comply with aliasing rules.
- Mark all code that accesses or manipulates data objects by independent input/output processes and independent, asynchronously interrupting processes as **VOLATILE**. For example, mark code which accesses shared variables and pointers to shared variables.
- Compile as much of your code as possible with **-O2**.
- If you encounter problems with **-O2**, check the code for any nonstandard use of aliasing rules before using the **-qalias=nostd** option.
- Next, use **-O3** on as much code as possible.
- If you encounter problems or performance degradations, consider using **-qstrict** or **-qcompact** along with **-O3** where necessary.
- If you still have problems with **-O3**, switch to **-O2** for a subset of files, but consider using **-qmaxmem=-1** or **-qnostrict**, or both.
- If your code works correctly with **-O2** or **-O3**, consider additionally specifying **-qhot** which can perform additional optimizations, especially on loops. Trying **-O3 -qhot** prior to trying **-O4** or **-O5** can improve your code without the compile-time overhead **-qipa** can have that is implied when using **-O4** and **-O5**.

The -O4 and -O5 Options

Optimization levels **-O4** and **-O5** automatically activate several other optimization options. Optimization level **-O4** includes:
- Everything from **-O3**
- **-qhot**
- **-qipa**
- **-qarch=auto**
- **-qtune=auto**

- **-qcache=auto**

Optimization level **-O5** includes:
- Everything from **-O4**
- **-qipa=level=2**

If **-O5** is specified on the compile step, then it should be specified on the link step, as well. Although the **-qipa** option is not strictly another optimization level, it extends the optimizations across procedures (even if the procedures are in different files). It enhances the effectiveness of the optimizations that are done by other optimization options, particularly **-O** (at any level) and **-Q**. Because it can also increase compile time substantially, you may want to use it primarily for tuning applications that are already debugged and ready to be used. If your application contains a mixture of Fortran and C or C++ code compiled with IBM XL C/C+ compilers, you can achieve additional optimization by compiling and linking all your code with the **-O5** option.

Optimizing for a Target Machine or Class of Machines

Target machine options are options that instruct the compiler to generate code for optimal execution on a given processor or architecture family. By default, the compiler generates code that runs on all supported systems, but perhaps suboptimally on a given system. By selecting appropriate target machine options, you can optimize your application to suit the broadest possible selection of target processors, a range of processors within a given family, or a specific processor. The following compiler options control optimizations affecting individual aspects of the target machine.

Target machine options

Option	Behavior
-q32	Generates code for a 32-bit addressing model (32-bit execution mode).
-q64	Generates code for a 64-bit addressing model (64-bit execution mode).
-qarch	Selects a family of processor architectures, or a specific architecture, for which instruction code should be generated.
-qtune	Biases optimization toward execution on a given processor, without implying anything about the instruction set architecture to use as a target.
-qcache	Defines a specific cache or memory geometry. The defaults are set through **-qtune**.

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

Related Information: See "-qarch Option" on page 127, "-qtune Option" on page 251, "-qcache Option" on page 137, and "Compiling for Specific Architectures" on page 39.

Getting the Most out of Target Machine Options

Try to specify with **-qarch** the smallest family of machines possible that will be expected to run your code reasonably well.

- **-qarch=auto** generates code that may take advantage of instructions available only on the compiling machine (or similar machines).
- To get sqrt optimization, you need to specify **-qarch=ppc64grsq** or another **-qarch** option that supports the square root instruction set.
- Specifying a **-qarch** option that is not compatible with your hardware, even though your program appears to work, may cause undefined behaviour; the compiler may emit instructions not available on that hardware.

Try to specify with **-qtune** the machine where performance should be best. If you are not sure, let the compiler determine how to best tune for optimization for a given **-qarch** setting.

Before using the **-qcache** option, look at the options sections of the listing using **-qlist** to see if the current settings are satisfactory. The settings appear in the listing itself when the **-qlistopt** option is specified. Modification of cache geometry may be useful in cases where the systems have configurable L2 or L3 cache options or where the execution mode reduces the effective size of a shared level of cache (for example, two-core-per-chip SMP execution on POWER4).

If you decide to use **-qcache**, use **-qhot** or **-qsmp** along with it.

Optimizing Floating-Point Calculations

Special compiler options exist for handling floating-point calculations efficiently. By default, the compiler makes a trade-off to violate certain IEEE floating-point rules in order to improve performance. For example, multiply-add instructions are generated by default because they are faster and produce a more precise result than separate multiply and add instructions. Floating-point exceptions, such as overflow or division by zero, are masked by default. If you need to catch these exceptions, you have the choice of enabling hardware trapping of these exceptions or using software-based checking. The option **-qflttrap** enables software-based checking. On the POWER4, POWER5, or PowerPC 970 processor, hardware trapping is recommended.

Options for handling floating-point calcluations

Option	Description
-qfloat	Provides precise control over the handling of floating-point calculations.
-qflttrap	Enables software checking of IEEE floating-point exceptions. This technique is sometimes more efficient than hardware checking because checks can be executed less frequently.

To understand the performance considerations for floating-point calculations with different combinations of compiler options, see "Maximizing Floating-Point Performance" on page 295 and "Minimizing the Performance Impact of Floating-Point Exception Trapping" on page 302.

High-order Transformations (-qhot)

High-order transformations are optimizations that specifically improve the performance of loops and array language. Optimization techniques can include interchange, fusion, and unrolling of loops, and reducing the generation of temporary arrays. The goals of these optimizations include:

- Reducing the costs of memory access through the effective use of caches and translation look-aside buffers.
- Overlapping computation and memory access through effective utilization of the data prefetching capabilities provided by the hardware.
- Improving the utilization of processor resources through reordering and balancing the usage of instructions with complementary resource requirements.

-qhot=vector is the default when **-qhot** is specified. Compiling with **-qhot=vector** transforms some loops to exploit optimized versions of functions rather than the standard versions. The optimized functions reside in a built-in library that includes functions and operations such as reciprocal, square root, and so on. The optimized versions make different trade-offs with respect to precision versus performance. Usage of **-qstrict** implies **-qhot=novector**.

Getting the Most out of -qhot

Try using **-qhot** along with **-O3** for all of your code. (The compiler assumes at least **-O2** level for **-qhot**.) It is designed to have a neutral effect when no opportunities for transformation exist.
- If you encounter unacceptably long compile times (this can happen with complex loop nests) or if your performance degrades with the use of **-qhot**, try using **-qhot=novector**, or **-qstrict** or **-qcompact** along with **-qhot**.
- If necessary, deactivate **-qhot** selectively, allowing it to improve some of your code.

Optimizing Loops and Array Language

The **-qhot** option does the following transformations to improve the performance of loops, array language, and memory management:
- Scalar replacement, loop blocking, distribution, fusion, interchange, reversal, skewing, and unrolling
- Reducing generation of temporary arrays

It requires at least level 2 of **-O**. The **-C** option inhibits it.

If you have SMP hardware, you can enable automatic parallelization of loops by specifying the **-qsmp** option. This optimization includes explicitly coded **DO** loops as well as **DO** loops that are generated by the compiler for array language (**WHERE**, **FORALL**, array assignment, and so on). The compiler can only parallelize loops that are independent (each iteration can be computed independently of any other iteration). One case where the compiler will not automatically parallelize loops is where the loops contain I/O, because doing so could lead to unexpected results. In this case, by using the **PARALLEL DO** or work-sharing **DO** directive, you can advise the compiler that such a loop can be safely parallelized. However, the type of I/O must be one of the following:
- Direct-access I/O where each iteration writes to or reads from a different record
- Sequential I/O where each iteration writes to or reads from a different unit
- Stream-access I/O where each iteration uses the **POS=** specifier to write to, or read from, a different part of the file.
- Stream-access I/O where each iteration writes to, or reads from, a different unit.

For more details, refer to the description of the **PARALLEL DO** or work-sharing **DO** directive in the *XL Fortran Enterprise Edition for AIX Language Reference*.

You can use the **-qhot** and **-qsmp** options on:
- Programs with performance bottlenecks that are caused by loops and structured memory accesses

- Programs that contain significant amounts of array language (which can be optimized in the same ways as FORTRAN 77 loops for array operations)

Cost Model for Loop Transformations

The loop transformations performed by the **-qhot** option are controlled by a set of assumptions about the characteristics of typical loops and the costs (in terms of registers used and potential delays introduced) of performing particular transformations.

The cost model takes into consideration:
- The number of available registers and functional units that the processor has
- The configuration of cache memory in the system
- The number of iterations of each loop
- The need to make conservative assumptions to ensure correct results

When the compiler can determine information precisely, such as the number of iterations of a loop, it uses this information to improve the accuracy of the cost model at that location in the program. If it cannot determine the information, the compiler relies on the default assumptions of the cost model. You can change these default assumptions, and thus influence how the compiler optimizes loops, by specifying compiler options:

- **-qassert=nodeps** asserts that none of the loops in the files being compiled have dependencies that extend from one iteration to any other iteration within the same loop. This is known as a loop-carried dependency. If you can assert that the computations performed during iteration *n* do not require results that are computed during any other iteration, the compiler is better able to rearrange the loops for efficiency.
- **-qassert=itercnt=***n* asserts that a "typical" loop in the files that you are compiling will iterate approximately *n* times. If this is not specified, the assumption is that loops iterate approximately 1024 times. The compiler uses this information to assist in transformations such as putting a high-iteration loop inside a low-iteration one.

  It is not crucial to get the value exactly right, and the value does not have to be accurate for every loop in the file. This value is not used if either of the following conditions is true:
  - The compiler can determine the exact iteration count.
  - You specified the **ASSERT(ITERCNT(***n***))** directive.

  Some of the loop transformations only speed up loops that iterate many times. For programs with many such loops or for programs whose hotspots and bottlenecks are high-iteration loops, specify a large value for *n*.

A program might contain a variety of loops, some of which are speeded up by these options and others unaffected or even slowed down. Therefore, you might want to determine which loops benefit most from which options, split some loops into different files, and compile the files with the set of options and directives that suits them best.

Unrolling Loops

Loop unrolling involves expanding the loop body to do the work of two, three, or more iterations, and reducing the iteration count proportionately. Benefits to loop unrolling include the following:

- Data dependence delays may be reduced or eliminated
- Loads and stores may be eliminated in successive loop iterations
- Loop overhead may be reduced

Loop unrolling also increases code sizes in the new loop body, which can increase register allocation and possibly cause register spilling. For this reason, unrolling sometimes does not improve performance.

Related Information: See "-qunroll Option" on page 255.

Describing the Hardware Configuration

The **-qtune** setting determines the default assumptions about the number of registers and functional units in the processor. For example, when tuning loops for execution on a Power2 architecture, **-qtune=pwr2** may cause the compiler to unroll most of the inner loops to a depth of two to take advantage of the extra arithmetic units.

The **-qcache** setting determines the blocking factor that the compiler uses when it blocks loops. The more cache memory that is available, the larger the blocking factor.

Efficiency of Different Array Forms

In general, operations on arrays with constant or adjustable bounds, assumed-size arrays, and pointee arrays require less processing than those on automatic, assumed-shape, or deferred-shape arrays and are thus likely to be faster.

Reducing Use of Temporary Arrays

If your program uses array language but never performs array assignments where the array on the left-hand side of the expression overlaps the array on the right-hand side, specifying the option **-qalias=noaryovrlp** can improve performance by reducing the use of temporary array objects.

The **-qhot** option can also eliminate many temporary arrays.

Array Padding

Because of the implementation of the XL Fortran-supported chip target cache architectures, array dimensions that are powers of 2 can lead to decreased cache utilization.

The optional **arraypad** suboption of the **-qhot** option permits the compiler to increase the dimensions of arrays where doing so might improve the efficiency of array-processing loops. If you have large arrays with some dimensions (particularly the first one) that are powers of 2 or if you find that your array-processing programs are slowed down by cache misses or page faults, consider specifying **-qhot=arraypad** or **-qhot=arraypad=***n* rather than just **-qhot**.

The padding that **-qhot=arraypad** performs is conservative. It also assumes that there are no cases in the source code (such as those created by an **EQUIVALENCE** statement) where storage elements have a relationship that is broken by padding. You can also manually pad array dimensions if you determine that doing so does not affect the program's results.

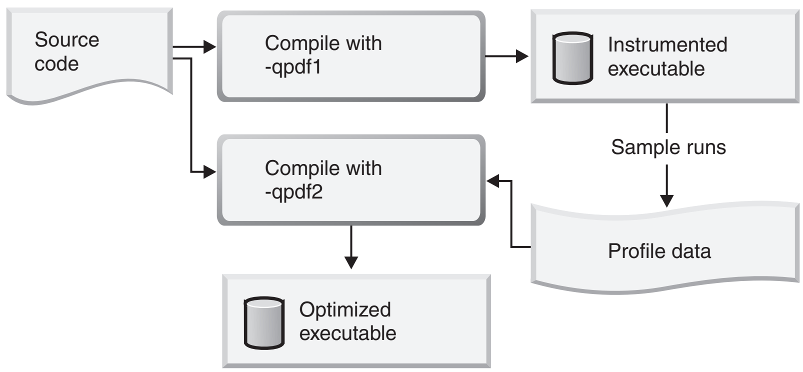
The additional storage taken up by the padding, especially for arrays with many dimensions, might increase the storage overhead of the program to the point where it slows down again or even runs out of storage. For more information, see "-qhot Option" on page 171.

Profile-directed Feedback (PDF)

Profile-directed feedback is a two-stage compilation process that lets you provide the compiler with data characteristic of typical program behavior. An instrumented executable is run in a number of different scenarios for an arbitrary amount of time, producing as a side effect a profile data file. A second compilation using the profile data produces an optimized executable.

PDF should be used mainly on code that has rarely executed conditional error handling or instrumentation. The technique has a neutral effect in the absence of firm profile information, but is not recommended if insufficient or uncharacteristic data is all that is available.

The following diagram illustrates the PDF process.



The two stages of the process are controlled by the compiler options **-qpdf1** and **-qpdf2**. Stage 1 is a regular compilation using an arbitrary set of optimization options and **-qpdf1**, that produces an executable or shared object that can be run in a number of different scenarios for an arbitrary amount of time. Stage 2 is a recompilation using the same options, except **-qpdf2** is used instead of **-qpdf1**, during which the compiler consumes previously collected data for the purpose of path-biased optimization.

Using Profile-directed Feedback (PDF)

You can optimize an application based on an analysis of how often it executes different sections of code, as follows:

1. Compile the application with **-qpdf1**.
2. Run the application using a typical data set or several typical data sets. When the application exits, it writes profiling information in the PDF file.
3. Compile the application with **-qpdf2**.

If you want to see which functions are used the most often, do the following:

1. Compile the application with **-qpdf1 -qshowpdf**.
2. Run the application using a typical data set or several typical data sets. The application writes more detailed profiling information in the PDF file.
3. Use the **showpdf** utility to view the information in the PDF file.

You can take more control of the PDF file generation, as follows:

1. Compile the application with **-qpdf1**.

2. Run the application using a typical data set or several typical data sets. This produces a PDF file in the current directory.
3. Copy the application to another directory and run it again. This produces a PDF file in the second directory.
4. Repeat the previous step as often as you want.
5. Use the **mergepdf** utility to combine the PDF files into one PDF file. For example, if you produce three PDF files that represent usage patterns that will occur 53%, 32%, and 15% of the time respectively, you can use this command:

```
mergepdf -r 53 path1  -r 32 path2  -r 15 path3
```

6. Compile the application with **-qpdf2**.

To erase the information in the PDF directory, use the **cleanpdf** or **resetpdf** utility.

Optimizing Conditional Branching

The **-qpdf** option helps to fine-tune the areas around conditional branches so that the default choices correspond to the most likely execution paths. Sometimes instructions from the more likely execution path run before the branch, in parallel with other instructions so that there is no slowdown.

Because the **-qpdf** option requires some extra compilation overhead and sample execution that uses representative data, you should use it mainly near the end of the development cycle.

Related Information: See "-qpdf Option" on page 210.

Interprocedural Analysis (-qipa)

Interprocedural analysis (IPA) enables the compiler to optimize across different files (whole-program analysis), and can result in significant performance improvements. Interprocedural analysis can be specified on the compile step only, or on both compile and link steps (***whole program*** mode). Whole program mode expands the scope of optimization to an entire program unit, which can be an executable or shared object. Whole program IPA analysis can consume significant amounts of memory and time when compiling or linking large programs.

IPA is enabled by the **-qipa** option. A summary of the effects of the most commonly used suboptions follows.

Commonly used **-qipa** suboptions

Suboption	Behavior
level=0	Program partioning and simple interprocedural optimization, which consists of: • Automatic recognition of standard libraries. • Localization of statically bound variables and procedures. • Partitioning and layout of procedures according to their calling relationships, which is also referred to as their *call affinity*. (Procedures that call each other frequently are located closer together in memory.) • Expansion of scope for some optimizations, notably register allocation.
level=1	Inlining and global data mapping. Specifically, • Procedure inlining.

- Partitioning and layout of static data according to reference affinity. (Data that is frequently referenced together will be located closer together in memory.)

This is the default level when **-qipa** is specified.

level=2
Global alias analysis, specialization, interprocedural data flow.
- Whole-program alias analysis. This level includes the disambiguation of pointer dereferences and indirect function calls, and the refinement of information about the side effects of a function call.
- Intensive intraprocedural optimizations. This can take the form of value numbering, code propagation and simplification, code motion into conditions or out of loops, elimination of redundancy.
- Interprocedural constant propagation, dead code elimination, pointer analysis.
- Procedure specialization (cloning).

inline=*inline-options*
Provides precise user control of inlining.

fine_tuning
Other values for **-qipa=** provide the ability to specify the behavior of library code, tune program partioning, read commands from a file, and so on.

Getting the Most from -qipa

It is not necessary to compile everything with **-qipa**, but try to apply it to as much of your program as possible. Here are some suggestions.
- When specifying optimization options in a makefile, remember to use the compiler command (**xlf**, **xlf90**, and so on) to link, and to include all compiler options on the link step.
- **-qipa** works when building executables or shared objects, but always compile main and exported functions with **-qipa**.
- When compiling and linking separately, use **-qipa=noobject** on the compile step for faster compilation.
- Ensure that there is enough space in **/tmp** (at least 200 MB), or use the TMPDIR environment variable to specify a different directory with sufficient free space.
- The **level** suboption is a throttle. Try varying it if link time is too long. Compiling with **-qipa=level=0** can be very beneficial for little additional link time.
- Look at the generated code after compiling with **-qlist** or **-qipa=list**. If too few or too many functions are inlined, consider using **-qipa=inline** or **-qipa=noinline**. To control inlining of a specific function, use **-Q+** and **-Q–**.
- If your application contains a mixture of Fortran and C or C++ code compiled with IBM XL C/C+ compilers, you can achieve additional optimization by compiling all your code with the **-qipa** option.

Optimizing Subprogram Calls

If a program has many subprogram calls, you can use the **-qipa=inline** option to turn on inlining, which reduces the overhead of such calls. Consider using the **-p** or **-pg** option with **prof** or **gprof**, respectively, to determine which subprograms are called most frequently and to list their names on the command line.

To make inlining apply to calls where the calling and called subprograms are in different scopes, include the **-qipa** option.

```
# Let the compiler decide (relatively cautiously) what to inline.
xlf95 -O3 -qipa=inline inline.f

# Encourage the compiler to inline particular subprograms.
xlf95 -O3 -qipa=inline=called_100_times,called_1000_times inline.f

# Explicity extend the inlining to calls across multiple files.
xlf95 -O3 -qipa=inline=called_100_times,called_1000_times -qipa inline.f
```

Related Information: See “-Q Option” on page 119 and “-qipa Option” on page 182.

Finding the Right Level of Inlining

Getting the right amount of inlining for a particular program may require some work on your part. The compiler has a number of safeguards and limits to avoid doing an excessive amount of inlining. Otherwise, it might perform less overall optimization because of storage constraints during compilation, or the resulting program might be much larger and run slower because of more frequent cache misses and page faults. However, these safeguards may prevent the compiler from inlining subprograms that you do want inlined. If this happens, you will need to do some analysis or rework or both to get the performance benefit.

As a general rule, consider identifying a few subprograms that are called most often, and inline only those subprograms.

Some common conditions that prevent **-qipa=inline** from inlining particular subprograms are:

- The calling and called procedures are in different scopes. If so, you can use the **-qipa** option to enable cross-file inlining.
- After the compiler has expanded a subprogram by a certain amount as a result of inlining, it does not inline subsequent calls from that subprogram. Again, there are different limits, which depend on whether the subprogram being called is named by a **-qipa=inline** option.

  Consider an example with three procedures: **A** is the caller, and **B** and **C** are at the upper size limit for automatic inlining. They are all in the same file, which is compiled like this:

  ```
  xlf -qipa=inline=c file.f
  ```

  The **-qipa=inline** means that calls to **C** are more likely to be inlined. If **B** and **C** were twice as large, calls to **B** would not be inlined at all, while some calls to **C** could still be inlined.

  Although these limits might prevent some calls from **A** to **B** or **A** to **C** from being inlined, the process starts over after the compiler finishes processing **A**.
- Any interface errors, such as different numbers, sizes, or types of arguments or return values, might prevent a call from being inlined. To locate such errors, compile with the **-qextchk** option, or define Fortran 90/Fortran 95 interface blocks for the procedures being called.
- Actual or potential aliasing of dummy arguments or automatic variables might prevent a procedure from being inlined. For example, inlining might not occur in the following cases:
  - If you compile the file containing either the calling or called procedure with the option **-qalias=nostd** and there are any arguments to the procedure being called
  - If there are more than approximately 31 arguments to the procedure being called

- If any automatic variables in the called procedures are involved in an **EQUIVALENCE** statement
- If the same variable argument is passed more than once in the same call: for example, **CALL SUB(X,Y,X)**

- Some procedures that use computed **GO TO** statements, where any of the corresponding statement labels are also used in an **ASSIGN** statement, might not be inlined.

To change the size limits that control inlining, you can specify **-qipa=limit=***n*, where *n* is 0 through 9. Larger values allow more inlining.

It is possible that C/C++ functions can be inlined into Fortran programs and visa-versa during link-time optimizations. The C/C++ code would have to be compiled using the IBM XL C/C++ compilers with **-qipa** and a compatible option set to that used in the XLF compilation.

Shared-memory Parallelism (-qsmp)

Some IBM processors are capable of shared-memory parallel processing. Compile with **-qsmp** to generate the threaded code needed to exploit this capability. The option implies a **-O2** optimization level. The default behavior for the option without suboptions is to do automatic parallelization with optimization.

The most commonly used **-qsmp** suboptions are summarized in the following table.

Commonly used **-qsmp** suboptions

Suboption	Behavior
auto	Instructs the compiler to automatically generate parallel code where possible without user assistance. This option also recognizes all the SMP directives.
omp	Enforces compliance with the OpenMP Fortran API for specifying explicit parallelism. Note that **-qsmp=omp** is currently incompatible with **-qsmp=auto**.
opt	Instructs the compiler to optimize as well as parallelize. The optimization is equivalent to **-O2 –qhot** in the absence of other optimization options. The default setting of **-qsmp** is **-qsmp=auto:noomp:opt**.
suboptions	Other values for the suboption provide control over thread scheduling, nested parallelism, locking, and so on.

Getting the Most out of -qsmp

- Use **-qsmp=omp:noauto** if you are compiling an OpenMP program and do not want automatic parallelization. By default, the parallelization performed is both explicit and automatic.
- Before using **-qsmp** with automatic parallelization, test your programs using optimization and **-qhot** in a single-threaded manner.
- Always use the reentrant compiler invocations (the **_r** command invocations, like **xlf_r**) when using **-qsmp**.
- By default, the runtime uses all available processors. Do not set the **XLSMPOPTS=PARTHDS** or **OMP_NUM_THREADS** variables unless you want to use fewer than the number of available processors. You might want to set the number of executing threads to a small number or to 1 to ease debugging.
- If you are using a dedicated machine or node, consider setting the **SPINS** and **YIELDS** variables (suboptions of **XLSMPOPTS**) to 0. Doing so prevents the operating system from intervening in the scheduling of threads across synchronization boundaries such as barriers.
- When debugging an OpenMP program, try using **-qsmp=noopt** (without **-O**) to make the debugging information produced by the compiler more precise. You

can also use the **SNAPSHOT** directive to create additional program points for storage visibility by flushing registers to memory.

Other Program Behavior Options

The precision of compiler analyses is significantly affected by instructions that can read or write memory. Aliasing pertains to alternate names for things, which in this context are references to memory. A reference to memory can be direct, as in the case of a named symbol, or indirect, as in the case of a pointer or dummy argument. A function call might also reference memory indirectly. Apparent references to memory that are false, that is, that do not actually reference some location assumed by the compiler, constitute barriers to compiler analysis.

Fortran defines a rule that dummy argument references may not overlap other dummy arguments or externally visible symbols during the execution of a subprogram.

The compiler performs sophisticated analyses, attempting to refine the set of possible aliases for pointer dereferences and calls. However, a limited scope and the absence of values at compile time constrain the effectiveness of these analyses. Increasing the optimization level, in particular, applying interprocedural analysis (that is, compiling with **-qipa**), can contribute to better aliasing.

Programs that violate language aliasing rules, as summarized above, commonly execute correctly without optimization or with low optimization levels, but can begin to fail when higher levels of optimization are attempted. The reason is that more aggressive optimizations take better advantage of aliasing information and can therefore expose subtly incorrect program semantics.

Options related to these issues are **-qstrict** and **-qalias**. Their behaviors are summarized in the table below.

Program behavior options

Option	Description
-qstrict, -qnostrict	Allows the compiler to reorder floating-point calculations and potentially excepting instructions. A potentially excepting instruction is one that may raise an interrupt due to erroneous execution (for example, floating-point overflow, a memory access violation). The default is **-qstrict** with **-qnoopt** and **-O2**; **-qnostrict** with **-O3**, **-O4**, and **-O5**.
-qalias	Allows the compiler to assume that certain variables do not refer to overlapping storage. The focus is on the overlap of dummy arguments and array assignments in Fortran.

Other Performance Options

Options are provided to control particular aspects of optimization. They are often enabled as a group or given default values when a more general optimization option is enabled.

Selected compiler options for optimizing performance

Option **Description**

-qcompact Chooses reduction of final code size over a reduction in execution time when a choice is necessary. Can be used to constrain **-O3** and higher optimizations.

-qsmallstack Instructs the compiler to limit the use of stack storage in the program. Doing so may increase heap usage.

-qunroll Independently controls loop unrolling. Is implicitly activated under **-O3** and higher optimizations.

-qtbtable Controls the generation of traceback table information.

-qunwind Informs the compiler that the stack can be unwound while a routine in this compilation is active. In other words, the compiler is informed that the application may or does rely on program stack unwinding mechanisms.

-qnounwind Informs the compiler that the stack will not be unwound while any routine in this compilation is active. The **-qnounwind** option enables optimization prologue tailoring, which reduces the number of saves and restores of nonvolatile registers.

-qlargepage Supports large 16M pages in addition to the default 4K pages, to allow hardware prefetching to be done more efficiently. Informs the compiler that heap and static data will be allocated from large pages at execution time.

Debugging Optimized Code

Debugging optimized programs presents special problems. Optimization may change the sequence of operations, add or remove code, and perform other transformations that make it difficult to associate the generated code with the original source statements. For example, the optimizer may remove all stores to a variable and keep it alive only in registers. Most debuggers are incapable of following this and it will appear as though that variable is never updated.

First debug your program, then recompile it with any optimization options, and test the optimized program before placing the program into production. If the optimized code does not produce the expected results, isolate the specific optimization problems in another debugging session.

The following table presents options that provide specialized information, which can be helpful during the development of optimized code.

Diagnostic options

Option	Behavior
-qlist	Instructs the compiler to emit an object listing. The object listing includes hex and pseudo-assembly representations of the generated instructions, traceback tables, and text constants.
-qreport	Instructs the compiler to produce a report of the loop transformations it performed and how the program was parallelized. The option is enabled when **-qhot** or **-qsmp** is specified.
-qinitauto	Instructs the compiler to emit code that initializes all automatic variables to a given value.
-qextchk	Generates additional symbolic information to allow the linker to do

cross-file type checking of external variables and functions. This option requires the linker **-btypchk** option to be active.

-qipa=list Instructs the compiler to emit an object listing that provides information for IPA optimization.

Different Results in Optimized Programs

Here are some reasons why an optimized program might produce different results from those of an unoptimized one:

- Optimized code can fail if a program contains code that is not valid. For example, failure can occur if the program passes an actual argument that also appears in a common block in the called procedure, or if two or more dummy arguments are associated with the same actual argument.
- If a program that worked without optimization fails when compiled with it, check the cross-reference listing and the execution flow of the program for variables that are used before they are initialized. Compile with the **-qinitauto=*hex_value*** option to try to produce the incorrect results consistently. For example, using **-qinitauto=FF** gives **REAL** and **COMPLEX** variables an initial value of ″negative not a number″ (-NAN). Any operations on these variables will also result in NAN values. Other bit patterns (***hex_value***) may yield different results and provide further clues as to what is going on. (Programs with uninitialized variables may appear to work properly when compiled without optimization, because of the default assumptions the compiler makes, but may fail when compiled with optimization. Similarly, a program may appear to execute correctly when optimized, but fails at lower optimization levels or when run in a different environment.)
- Use with caution debugging techniques that rely on examining values in storage. The compiler might have deleted or moved a common expression evaluation. It might have assigned some variables to registers, so that they do not appear in storage at all.

Related Information: See “-g Option” on page 108, “-qinitauto Option” on page 177, and “Problem Determination and Debugging” on page 369.

Compiler-friendly Programming

Compiler-friendly programming idioms can be as useful to performance as any of the options or directives. Here are some suggestions.

General

- Where possible, use command invocations like **xlf90** or **xlf95** to enhance standards conformance and code portability. If this is not possible, consider using the **-qnosave** option to have all local variables be automatic; doing this provides more opportunities for optimization.
- Use modules to group related subroutines and functions.
- Consider using the highly tuned **MASS** and **ESSL** libraries rather than custom implementations or generic libraries.

Hand-tuning

- Do not excessively hand-optimize your code. Unusual constructs can confuse the compiler (and other programmers), and make your application difficult to optimize for new machines.
- Do limited hand tuning of small functions by inlining.

- Avoid breaking your program into too many small functions as this can increase the percentage of time the program spends in dealing with call overhead. If you choose to use many small functions, seriously consider using **-qipa**.

Variables

- Avoid unnecessary use of global variables and pointers. When using them in a loop, load them into a local variable before the loop and store them back after.
- Use the **INTENT** statement to describe usage of parameters.

Conserving storage

- Use register-sized integers (INTEGER(4) or INTEGER(8) data type) for scalars.
- Use the smallest floating-point precision appropriate to your computation. Use the **REAL(16)**, or **COMPLEX(32)** data type only when extremely high precision is required.
- When writing new code, use module variables rather than common blocks for global storage.
- Use the **CONTAINS** statement only to share thread local storage.

Pointers

- Obey all language aliasing rules. Try to avoid using **–qalias=nostd**.
- Limit the use of **ALLOCATABLE** arrays and **POINTER** variables to situations which demand dynamic allocation.

Arrays

- Use local variables wherever possible for loop index variables and bounds.
- Keep array index expressions as simple as possible. Where indexing needs to be indirect, consider using the **PERMUTATION** directive.
- When using array assignment or **WHERE** statements, pay close attention to the generated code with **-qlist** or **-qreport**. If performance is inadequate, consider using -qhot or rewriting array language in loop form.

Implementation Details of XL Fortran Input/Output

This section discusses XL Fortran support (through extensions and platform-specific details) for the AIX file system.

Related Information: See “-qposition Option” on page 219 and “Mixed-Language Input and Output” on page 346.

Implementation Details of File Formats

XL Fortran implements files in the following manner:

Sequential-access unformatted files:
An integer that contains the length of the record precedes and follows each record. The length of the integer is 4 bytes for 32-bit applications. It is 4 bytes if you set the **uwidth** run-time option to 32 (the default) and it is 8 bytes if you set **uwidth** to 64 for 64-bit applications.

Sequential-access formatted files:
XL Fortran programs break these files into records while reading, by using each newline character (X'0A') as a record separator.

On output, the input/output system writes a newline character at the end of each record. Programs can also write newline characters for themselves. This practice is not recommended because the effect is that the single record that appears to be written is treated as more than one record when being read or backspaced over.

Direct access files:
XL Fortran simulates direct-access files with files whose length is a multiple of the record length of the XL Fortran file. You must specify, in an **OPEN** statement, the record length (**RECL**) of the direct-access file. XL Fortran uses this record length to distinguish records from each other.

For example, the third record of a direct-access file of record length 100 bytes would start at the 201st byte of the single record of an AIX file and end at the 300th byte.

If the length of the record of a direct-access file is greater than the total amount of data you want to write to the record, XL Fortran pads the record on the right with blanks (X'20').

Stream-access unformatted files:
Unformatted stream files are viewed as a collection of file storage units. In XL Fortran, a file storage unit is one byte.

A file connected for unformatted stream access has the following properties:

- The first file storage unit has position 1. Each subsequent file storage unit has a position that is one greater than that of the preceding one.
- For a file that can be positioned, file storage units need not be read or written in the order of their position. Any file storage unit may be read from the file while it is connected to a unit, provided that the file storage unit has been written since the file was created, and if a READ statement for the connection is permitted.

Stream-access formatted files:

A record file connected for formatted stream access has the following properties:

- Some file storage units may represent record markers. The record marker is the newline character (X'0A').
- The file will have a record structure in addition to the stream structure.
- The record structure is inferred from the record markers that are stored in the file.
- Records can have any length up to the internal limit allowed by XL Fortran (See Appendix D, "XL Fortran Internal Limits," on page 425.)
- There may or may not be a record marker at the end of the file. If there is no record marker at the end of the file, the final record is incomplete, but not empty.

A file connected for formatted stream access has the following properties:

- The first file storage unit has position 1. Each subsequent file storage unit has a position that is greater than that of the preceding one. Unlike unformatted stream access, the positions of successive file storage units are not always consecutive.
- The position of a file connected for formatted stream access can be determined by the **POS=** specifier in an **INQUIRE** statement.
- For a file that can be positioned, the file position can be set to a value that was previously identified by the **POS=** specifier in **INQUIRE**.

File Names

You can specify file names as either relative (such as **file**, **dir/file**, or **../file**) or absolute (such as **/file** or **/dir/file**). The maximum length of a file name (the full path name) is 1023 characters, even if you only specify a relative path name in the I/O statement. The maximum length of a file name with no path is 255 characters.

You must specify a valid file name in such places as the following:

- The **FILE=** specifier of the **OPEN** and **INQUIRE** statements
- **INCLUDE** lines

Related Information: To specify a file whose location depends on an environment variable, you can use the **GETENV** intrinsic procedure to retrieve the value of the environment variable:

```
character(100) home, name
call getenv('HOME', value=home)
! Now home = $HOME + blank padding.
! Construct the complete path name and open the file.
name=trim(home) // '/remainder/of/path'
open (unit=10, file=name)
...
end
```

Preconnected and Implicitly Connected Files

Units 0, 5, and 6 are preconnected to standard error, standard input, and standard output, respectively, before the program runs.

All other units can be implicitly connected when an **ENDFILE**, **PRINT**, **READ**, **REWIND**, or **WRITE** statement is performed on a unit that has not been opened. Unit *n* is connected to a file that is named **fort**.*n*. These files need not exist, and XL Fortran does not create them unless you use their units.

Note: Because unit 0 is preconnected for standard error, you cannot use it for the following statements: **CLOSE**, **ENDFILE**, **BACKSPACE**, **REWIND**, and direct or stream input/output. You can use it in an **OPEN** statement only to change the values of the **BLANK=**, **DELIM=**, or **PAD=** specifiers.

You can also implicitly connect units 5 and 6 (and *) by using I/O statements that follow a **CLOSE**:

```
      WRITE (6,10) "This message goes to stdout."
      CLOSE (6)
      WRITE (6,10) "This message goes in the file fort.6."
      PRINT *, "Output to * now also goes in fort.6."
10    FORMAT (A)
      END
```

The **FORM=** specifier of implicitly connected files has the value **FORMATTED** before any **READ**, **WRITE**, or **PRINT** statement is performed on the unit. The first such statement on such a file determines the **FORM=** specifier from that point on: **FORMATTED** if the formatting of the statement is format-directed, list-directed, or namelist; and **UNFORMATTED** if the statement is unformatted.

Preconnected files also have **FORM='FORMATTED'**, **STATUS='OLD'**, and **ACTION='READWRITE'** as default specifier values.

The other properties of a preconnected or implicitly connected file are the default specifier values for the **OPEN** statement. These files always use sequential access.

If you want XL Fortran to use your own file instead of the **fort.*n*** file, you can either specify your file for that unit through an **OPEN** statement or create a symbolic link before running the application. In the following example, there is a symbolic link between **myfile** and **fort.10**:

```
ln myfile fort.10
```

When you run an application that uses the preconnected file **fort.10** for input/output, XL Fortran uses the file **myfile** instead. The file **fort.10** exists, but only as a symbolic link. The following command will remove the symbolic link, but will not affect the existence of **myfile**:

```
rm fort.10
```

File Positioning

Table 21. Position of the File Pointer When a File Is Opened with No POSITION= Specifier

-qposition suboptions	Implicit OPEN		Explicit OPEN					
			STATUS = 'NEW'		STATUS = 'OLD'		STATUS = 'UNKNOWN'	
	File exists	File does not exist	File exists	File does not exist	File exists	File does not exist	File exists	File does not exist
option not specified	Start	Start	Error	Start	Start 1, 3	Error	Start	Start
appendold 2	Start	Start	Error	Start	End	Error	Start	Start
appendunknown	Start	Start	Error	Start	Start 3	Error	End	Start
appendold and appendunknown	Start	Start	Error	Start	End	Error	End	Start

The important things to note are:

1 The behavior of the **xlf90**, **xlf90_r**, **xlf90_r7**, **xlf95**, **xlf95_r**, and **xlf95_r7** commands when you do not specify an option is different from XL Fortran Version 2.3 in this case. The Fortran 90 and Fortran 95 standards require this behavior. To minimize migration problems, the **xlf**, **xlf_r**, **xlf_r7**, **f77**, and **fort77** commands keep the same default as XL Fortran Version 2.3 and append to the end of the file.

Attention: If your program depends on the old behavior to append to the end of an existing file with **STATUS='OLD'**, you need to use the option **-qposition=appendold** or **POSITION=** specifiers when making the switch to the **xlf90**, **xlf90_r**, **xlf90_r7**, **xlf95**, **xlf95_r**, or **xlf95_r7** command. Otherwise, when you compile the program with these commands and run it, the new data will overwrite the file instead of appending to it.

2 **-qposition=appendold** produces the default XL Fortran Version 2.3 behavior for positioning the file pointer. This option is in the configuration-file stanza for the **xlf**, **xlf_r**, **xlf_r7**, **f77**, and **fort77** commands but is not in the configuration-file stanza for the **xlf90**, **xlf90_r**, **xlf90_r7**, **xlf95**, **xlf95_r**, and **xlf95_r7** commands.

3 This file position was not possible in XL Fortran Version 2.3.

Preserving the XL Fortran Version 2.3 File Positioning

If you are upgrading from XL Fortran Version 2.3 and want the file positioning to work the same way as before, note the following guidelines:

- As long as you continue to use the **xlf_r**, **xlf_r7**, **xlf**, **f77**, and **fort77** commands, you do not need to make any changes.
- When you make the transition to the **xlf90**, **xlf90_r**, **xlf90_r7**, **xlf95**, **xlf95_r**, and **xlf95_r7** commands:
  - Add **-qposition=appendold** for programs that were previously compiled without any **-qposition** option.
  - Add **-qposition=appendold:appendunknown** for programs that were previously compiled with **-qposition=append**.

I/O Redirection

You can use the redirection operator on the command line to redirect input to and output from your XL Fortran program. How you specify and use this operator depends on which shell you are running. Here is a **ksh** example:

```
$ cat redirect.f
       write (6,*) 'This goes to standard output'
       write (0,*) 'This goes to standard error'
       read (5,*) i
       print *,i
       end
$ xlf95 redirect.f
** _main    === End of Compilation 1 ===
1501-510  Compilation successful for file redirect.f.
$ # No redirection. Input comes from the terminal. Output goes to
$ # the screen.
$ a.out
 This goes to standard output
 This goes to standard error
4
 4
$ # Create an input file.
$ echo >stdin 2
$ # Redirect each standard I/O stream.
$ a.out >stdout 2>stderr <stdin
$ cat stdout
 This goes to standard output
 2
$ cat stderr
 This goes to standard error
```

You can refer to the following sections of the *AIX Commands Reference* for more information on redirection:
- “Input and Output Redirection in the Korn Shell (ksh Command)”
- “Input and Output Redirection in the Bourne Shell (bsh Command)”
- “Input and Output Redirection in the C Shell (csh Command)”

How XLF I/O Interacts with Pipes, Special Files, and Links

You can access regular operating system files and blocked special files by using sequential-access, direct-access, or stream-access methods.

You can only access pseudo-devices, pipes, and character special files by using sequential-access methods, or stream-access without using the **POS=** specifier.

When you link files together, you can use their names interchangeably, as shown in the following example:

```
OPEN (4, FILE="file1")
OPEN (4, FILE="link_to_file1", PAD="NO") ! Modify connection
```

Do not specify the **POSITION=** specifier as **REWIND** or **APPEND** for pipes. **REWIND** is allowed for tapes, but **APPEND** is not. To open a tape file at a specific location, use the **tctl** command to position the tape before running the Fortran program, and specify **POSITION=’ASIS’** in the program.

Do not specify **ACTION=’READWRITE’** for a pipe.

Do not use the **BACKSPACE** statement on files that are pseudo-devices or character special files (such as tapes).

Do not use the **REWIND** statement on files that are pseudo-devices or pipes. If used on a tape, it rewinds to the beginning of the file, not the beginning of the tape.

Default Record Lengths

If a pseudo-device, pipe, or character special file is connected for formatted or unformatted sequential access with no **RECL=** specifier, or for formatted stream access, the default record length is 32 768 rather than 2 147 483 647, which is the default for sequential-access files connected to random-access devices. (See the **default_recl** run-time option.)

In certain cases, the default maximum record length for formatted files is larger, to accommodate programs that write long records to standard output. If a unit is connected to a terminal for formatted sequential access and there is no explicit **RECL=** qualifier in the **OPEN** statement, the program uses a maximum record length of 2 147 483 646 (2**31-2) bytes, rather than the usual default of 32 768 bytes. When the maximum record length is larger, formatted I/O has one restriction: **WRITE** statements that use the **T** or **TL** edit descriptors must not write more than 32 768 bytes. This is because the unit's internal buffer is flushed each 32 768 bytes, and the **T** or **TL** edit descriptors will not be able to move back past this boundary.

File Permissions

A file must have the appropriate permissions (read, write, or both) for the corresponding operation being performed on it.

When a file is created, the default permissions (if the **umask** setting is 000) are both read and write for user, group, and other. You can turn off individual permission bits by changing the **umask** setting before you run the program.

Selecting Error Messages and Recovery Actions

By default, an XLF-compiled program continues after encountering many kinds of errors, even if the statements have no **ERR=** or **IOSTAT=** specifiers. The program performs some action that might allow it to recover successfully from the bad data or other problem.

To control the behavior of a program that encounters errors, set the **XLFRTEOPTS** environment variable, which is described in "Setting Run-Time Options" on page 51, before running the program:

- To make the program stop when it encounters an error instead of performing a recovery action, include **err_recovery=no** in the **XLFRTEOPTS** setting.
- To make the program stop issuing messages each time it encounters an error, include **xrf_messages=no**.
- To disallow XL Fortran extensions to Fortran 90 at run time, include **langlvl=90std**. To disallow XL Fortran extensions to Fortran 95 at run time, include **langlvl=95std**. To disallow XL Fortran extensions to Fortran 2003 behaviour at run time, include **langlvl=2003std**. These settings, in conjunction with the **-qlanglvl** compiler option, can help you locate extensions when preparing to port a program to another platform.

For example:

```
# Switch defaults for some run-time settings.
XLFRTEOPTS="err_recovery=no:cnverr=no"
export XLFRTEOPTS
```

If you want a program always to work the same way, regardless of environment-variable settings, or want to change the behavior in different parts of the program, you can call the **SETRTEOPTS** procedure:

```
PROGRAM RTEOPTS
USE XLFUTILITY
CALL SETRTEOPTS("err_recovery=no") ! Change setting.
... some I/O statements ...
CALL SETRTEOPTS("err_recovery=yes") ! Change it back.
... some more I/O statements ...
END
```

Because a user can change these settings through the **XLFRTEOPTS** environment variable, be sure to use **SETRTEOPTS** to set all the run-time options that might affect the desired operation of the program.

Flushing I/O Buffers

To protect data from being lost if a program ends unexpectedly, you can use the **FLUSH** statement or the **flush_** subroutine to write any buffered data to a file. (The **FLUSH** statement is recommended for better portability.) The following example shows use of the **flush_** subroutine:

```
USE XLFUTILITY
PARAMETER (UNIT=10)

DO I=1,1000000
    WRITE (10,*) I
    CALL MIGHT_CRASH
! If the program ends in the middle of the loop, some data
! may be lost.
END DO

DO I=1,1000000
    WRITE (10,*) I
    CALL FLUSH_(UNIT)
    CALL MIGHT_CRASH
! If the program ends in the middle of the loop, all data written
! up to that point will be safely in the file.
END DO

END
```

Related Information: See "Mixed-Language Input and Output" on page 346 and the **FLUSH** statement in the *XL Fortran Enterprise Edition for AIX Language Reference*.

Choosing Locations and Names for Input/Output Files

If you need to override the default locations and names for input/output files, you can use the following methods without making any changes to the source code.

Naming Files That Are Connected with No Explicit Name

To give a specific name to a file that would usually have a name of the form **fort.***unit*, you must set the run-time option **unit_vars** and then set an environment variable with a name of the form **XLFUNIT_***unit* for each scratch file. The association is between a unit number in the Fortran program and a path name in the file system.

For example, suppose that the Fortran program contains the following statements:

```
OPEN (UNIT=1, FORM='FORMATTED', ACCESS='SEQUENTIAL', RECL=1024)
...
OPEN (UNIT=12, FORM='UNFORMATTED', ACCESS='DIRECT', RECL=131072)
...
OPEN (UNIT=123, FORM='UNFORMATTED', ACCESS='SEQUENTIAL', RECL=997)
```

```
XLFRTEOPTS="unit_vars=yes"     # Allow overriding default names.
XLFUNIT_1="/tmp/molecules.dat" # Use this named file.
XLFUNIT_12="../data/scratch"   # Relative to current directory.
XLFUNIT_123="/home/user/data"  # Somewhere besides /tmp.
export XLFRTEOPTS XLFUNIT_1 XLFUNIT_12 XLFUNIT_123
```

Notes:

1. The **XLFUNIT_***number* variable name must be in uppercase, and *number* must not have any leading zeros.
2. **unit_vars=yes** might be only part of the value for the **XLFRTEOPTS** variable, depending on what other run-time options you have set. See "Setting Run-Time Options" on page 51 for other options that might be part of the **XLFRTEOPTS** value.
3. If the **unit_vars** run-time option is set to **no** or is undefined or if the applicable **XLFUNIT_***number* variable is not set when the program is run, the program uses a default name (**fort.***unit*) for the file and puts it in the current directory.

Naming Scratch Files

To place all scratch files in a particular directory, set the **TMPDIR** environment variable to the name of the directory. The program then opens the scratch files in this directory. You might need to do this if your **/tmp** directory is too small to hold the scratch files.

To give a specific name to a scratch file, you must do the following:

1. Set the run-time option **scratch_vars**.
2. Set an environment variable with a name of the form **XLFSCRATCH_***unit* for each scratch file.

The association is between a unit number in the Fortran program and a path name in the file system. In this case, the **TMPDIR** variable does not affect the location of the scratch file.

For example, suppose that the Fortran program contains the following statements:

```
      OPEN (UNIT=1, STATUS='SCRATCH', &
            FORM='FORMATTED', ACCESS='SEQUENTIAL', RECL=1024)
      ...
      OPEN (UNIT=12, STATUS='SCRATCH', &
            FORM='UNFORMATTED', ACCESS='DIRECT', RECL=131072)
      ...
      OPEN (UNIT=123, STATUS='SCRATCH', &
            FORM='UNFORMATTED', ACCESS='SEQUENTIAL', RECL=997)
```

```
XLFRTEOPTS="scratch_vars=yes"     # Turn on scratch file naming.
XLFSCRATCH_1="/tmp/molecules.dat" # Use this named file.
XLFSCRATCH_12="../data/scratch"   # Relative to current directory.
XLFSCRATCH_123="/home/user/data"  # Somewhere besides /tmp.
export XLFRTEOPTS XLFSCRATCH_1 XLFSCRATCH_12 XLFSCRATCH_123
```

Notes:

1. The **XLFSCRATCH_***number* variable name must be in uppercase, and *number* must not have any leading zeros.
2. **scratch_vars=yes** might be only part of the value for the **XLFRTEOPTS** variable, depending on what other run-time options you have set. See "Setting Run-Time Options" on page 51 for other options that might be part of the **XLFRTEOPTS** value.
3. If the **scratch_vars** run-time option is set to **no** or is undefined or if the applicable **XLFSCRATCH_***number* variable is not set when the program is run, the program chooses a unique file name for the scratch file and puts it in the directory named by the **TMPDIR** variable or in the **/tmp** directory if the **TMPDIR** variable is not set.

Increasing Throughput with Logical Volume I/O and Data Striping

For performance-critical applications, the overhead of the Journaled File System (JFS) for I/O operations might slow down the program. If your program generates large scratch files, you might find that I/O bandwidth also limits its performance. Performing I/O directly to a logical volume rather than to a file system can eliminate the JFS overhead. Using data striping on the logical volume can further improve throughput or processor utilization or both.

Related Information: Because data-striped I/O runs much faster for data items that are aligned more strictly than normal, be sure to use the "-qalign Option" on page 125 when compiling any programs that perform logical volume I/O or data striping.

Logical Volume I/O

To use a logical volume as a file, do the following:

- Set up the logical volume with permissions that allow you to read or write it.
- Specify the name of the special file (for example, **/dev/rlv99**) in the **OPEN** statement.

Attention: Do not perform this kind of I/O with any logical volume that already contains a file system; doing so will destroy the file system. You must also take any precautions necessary to ensure that multiple users or programs do not write to the same logical volume or write to a logical volume while someone else is reading from it.

Notes:

1. A logical volume can only be opened as a single direct-access file with a record length that is a multiple of the logical volume's sector size (usually 512 bytes).
2. I/O operations are not guaranteed to detect attempts to read or write past the end of the logical volume. Therefore, make sure that the program keeps track of the extent of the logical volume. The maximum amount of data that can be stored this way on logical volume is the size of the logical volume minus the size of one stripe. The XL Fortran I/O routines use this stripe for bookkeeping.
3. For optimal performance of data striping, ensure that any data items that you specified in the read or write lists for a logical volume are aligned on 64-byte boundaries. The simplest way to ensure this alignment for large static arrays and common blocks is to specify the option **-qalign=4k**.
4. Regardless of any **STATUS='SCRATCH'** or **STATUS='DELETE'** specifiers, neither the data in a logical volume nor the special file in **/dev** is destroyed by an **OPEN** or **CLOSE** statement.

Related Information: See "-qalign Option" on page 125.

Data Striping

Data striping is primarily useful for increasing I/O throughput for large, direct-access scratch files. The performance benefit is greatest when a program reads and writes large objects.

When you make use of data striping, you perform I/O to a logical volume as described in "Logical Volume I/O" and set up the logical volume especially for high-performance striped I/O through the **smit** or **mklv** commands. You can then use the technique that is described in "Naming Scratch Files" on page 332 to place a scratch file on a striped logical volume.

For example, consider a Fortran program that contains the following statements:

```
 OPEN (UNIT=42, STATUS='SCRATCH',
+      FORM='UNFORMATTED', ACCESS='DIRECT', RECL=131072)
 ...
 OPEN (UNIT=101, STATUS='SCRATCH',
+      FORM='UNFORMATTED', ACCESS='DIRECT', RECL=131072)
```

You could place the scratch files for units 42 and 101 on the raw logical volumes **/dev/rlv30** and **/dev/rlv31** by setting environment variables before running the program, as follows:

```
XLFRTEOPTS="scratch_vars=yes"
XLFSCRATCH_42="/dev/rlv30"
XLFSCRATCH_101="/dev/rlv31"
export XLFRTEOPTS XLFSCRATCH_42 XLFSCRATCH_101
```

Related Information: *AIX Performance Management Guide* discusses the performance of data striping.

Asynchronous I/O

You may need to use asynchronous I/O for speed and efficiency in scientific programs that perform I/O for large amounts of data. Synchronous I/O blocks the execution of an application until the I/O operation completes. Asynchronous I/O allows an application to continue processing while the I/O operation is performed in the background. You can modify applications to take advantage of the ability to overlap processing and I/O operations. Multiple asynchronous I/O operations can also be performed simultaneously on multiple files that reside on independent devices. For a complete description of the syntax and language elements that you require to use this feature, see the *XL Fortran Enterprise Edition for AIX Language Reference* under the topics:

- **INQUIRE** Statement
- **READ** Statement
- **WAIT** Statement
- **WRITE** Statement

Execution of an Asychronous Data Transfer Operation

The effect of executing an asynchronous data transfer operation will be as if the following steps were performed in the order specified, with steps (6)-(9) possibly occurring asynchronously:

1. Determine the direction of the data transfer.
2. Identify the unit.
3. Establish the format if one is present.
4. Determine whether an error condition, end-of-file condition, or end-of-record condition has occurred.
5. Cause the variable that you specified in the **IOSTAT=** specifier in the data transfer statement to become defined.
6. Position the file before you transfer data.
7. Transfer data between the file and the entities that you specified by the input/output list (if any).
8. Determine whether an error condition, end-of-file condition, or end-of-record condition has occurred.
9. Position the file after you transfer data.
10. Cause any variables that you specified in the **IOSTAT=** and **SIZE=** specifiers in the **WAIT** statement to become defined.

Usage

You can use Fortran asynchronous **READ** and **WRITE** statements to initiate asynchronous data transfers in Fortran. Execution continues after the asynchronous I/O statement, regardless of whether the actual data transfer has completed.

A program may synchronize itself with a previously initiated asynchronous I/O statement by using a **WAIT** statement. There are two forms of the **WAIT** statement:

1. In a **WAIT** statement without the **DONE=** specifier, the **WAIT** statement halts execution until the corresponding asynchronous I/O statement has completed:

```
integer idvar
integer, dimension(1000):: a
....
```

```
READ(unit_number,ID=idvar) a
....
WAIT(ID=idvar)
....
```

2. In a **WAIT** statement with the **DONE=** specifier, the **WAIT** statement returns the completion status of an asynchronous I/O statement:

```
integer idvar
logical done
integer, dimension(1000):: a
....
READ(unit_number,ID=idvar) a
....
WAIT(ID=idvar, DONE=done)
....
```

   The variable you specified in the **DONE=** specifier is set to ″true″ if the corresponding asynchronous I/O statement completes. Otherwise, it is set to ″false″.

The actual data transfer can take place in the following cases:
- During the asynchronous **READ** or **WRITE** statement
- At any time before the execution of the corresponding **WAIT** statement
- During the corresponding **WAIT** statement

Because of the nature of asynchronous I/O, the actual completion time of the request cannot be predicted.

You specify Fortran asynchronous **READ** and **WRITE** statements by using the **ID=** specifier. The value set for the **ID=** specifier by an asynchronous **READ** or **WRITE** statement must be the same value specified in the **ID=** specifier in the corresponding **WAIT** statement. You must preserve this value until the associated asynchronous I/O statement has completed.

The following program shows a valid asynchronous **WRITE** statement:

```
program sample0
integer, dimension(1000):: a
integer idvar
a = (/(i,i=1,1000)/)
WRITE(10,ID=idvar) a
WAIT(ID=idvar)
end
```

The following program is not valid, because XL Fortran destroys the value of the asynchronous I/O identifier before the associated **WAIT** statement:

```
program sample1
integer, dimension(1000):: a
integer idvar
a = (/(i,i=1,1000)/)
WRITE(10,ID=idvar) a
idvar = 999     ! Valid id is destroyed.
WAIT(ID=idvar)
end
```

An application that uses asynchronous I/O typically improves performance by overlapping processing with I/O operations. The following is a simple example:

```
program sample2
integer  (kind=4), parameter :: isize=1000000, icol=5
integer  (kind=4) :: i, j, k
integer  (kind=4), dimension(icol) :: handle
```

```
       integer  (kind=4), dimension(isize,icol), static :: a, a1

!
!      Opens the file for both synchronous and asynchronous I/O.
!
       open(20,form="unformatted",access="direct", &
          status="scratch", recl=isize*4,asynch="yes")

!
!      This loop overlaps the initialization of a(:,j) with
!      asynchronous write statements.
!
!      NOTE: The array is written out one column at a time.
!            Since the arrays in Fortran are arranged in column
!            major order, each WRITE statement writes out a
!            contiguous block of the array.
!
       do 200 j = 1, icol
          a(:,j) = (/ (i*j,i=1,isize) /)
          write(20, id=handle(j), rec=j) a(:,j)
200    end do

!
!      Wait for all writes to complete before reading.
!
       do 300 j = 1, icol
          wait(id=handle(j))
300    end do

!
!      Reads in the first record.
!
       read(20, id=handle(1), rec=1) a1(:,1)

       do 400 j = 2, icol
          k = j - 1
!
!         Waits for a previously initiated read to complete.
!
          wait(id=handle(k))
!
!         Initiates the next read immediately.
!
          read(20, id=handle(j), rec=j) a1(:,j)
!
!         While the next read is going on, we do some processing here.
!
          do 350 i = 1, isize
             if (a(i,k) .ne. a1(i,k)) then
                print *, "(",i,",",k,") &
                &  expected ", a(i,k), " got ", a1(i,k)
             end if
350       end do
400    end do

!
!      Finish the last record.
!
       wait(id=handle(icol))

       do 450 i = 1, isize
          if (a(i,icol) .ne. a1(i,icol)) then
             print *, "(",i,",",icol,") &
             &  expected ", a(i,icol), " got ", a1(i,icol)
          end if
```

```
450     end do

        close(20)
        end
```

Performance

To maximize the benefits of asynchronous I/O, you should only use it for large contiguous data items.

It is possible to perform asynchronous I/O on a large number of small items, but the overall performance will suffer. This is because extra processing overhead is required to maintain each item for asynchronous I/O. Performing asynchronous I/O on a larger number of small items is strongly discouraged. The following are two examples:

1. WRITE(unit_number, ID=idvar) a1(1:100000000:2)
2. WRITE(unit_number, ID=idvar) (a2(i,j),j=1,100000000)

Performing asynchronous I/O on unformatted sequential files is less efficient. This is because each record might have a different length, and these lengths are stored with the records themselves. You should use unformatted direct access or unformatted stream access, if possible, to maximize the benefits of asynchronous I/O.

Compiler-Generated Temporary I/O Items

There are situations when the compiler must generate a temporary variable to hold the result of an I/O item expression. In such cases, synchronous I/O is performed on the temporary variable, regardless of the mode of transfer that you specified in the I/O statement. The following are examples of such cases:

1. For **READ**, when an array with vector subscripts appears as an input item:
   a.
   ```
         integer a(5), b(3)

         b = (/1,3,5/)
         read(99, id=i) a(b)
   ```
   b.
   ```
         real a(10)
         read(99,id=i) a((/1,3,5/))
   ```
2. For **WRITE**, when an output item is an expression that is a constant or a constant of certain derived types:
   a.
   ```
      write(99,id=i) 1000
   ```
   b.
   ```
      integer a
      parameter(a=1000)

      write(99,id=i) a
   ```

c.
```
type mytype
integer a
integer b
end type mytype

write(99,id=i) mytype(4,5)
```

3. For **WRITE**, when an output item is a temporary variable:
   a.
   ```
   write(99,id=i) 99+100
   ```
   b.
   ```
   write(99,id=i) a+b
   ```
   c.
   ```
   external ff
   real(8) ff

   write(99,id=i) ff()
   ```
4. For **WRITE**, when an output item is an expression that is an array constructor:
   ```
   write(99,id=i) (/1,2,3,4,5/)
   ```
5. For **WRITE**, when an output item is an expression that is a scalarized array:
   ```
   integer a(5),b(5)
   write(99,id=i) a+b
   ```

System Setup

Before a Fortran application that is using asynchronous I/O can run on an AIX system, you must enable AIX asynchronous I/O. If you did not enable AIX asynchronous I/O, a Fortran program using asynchronous I/O statements cannot be loaded. This will result in the following messages being displayed:

```
Could not load program asyncio
Symbol kaio_rdwr in ksh is undefined
Symbol listio in ksh is undefined
Symbol acancel in ksh is undefined
Symbol iosuspend in ksh is undefined
Error was: Exec format error
```

For information on how to configure your system for asynchronous I/O, see "Changing Attributes for Asynchronous I/O" in *AIX Version 4 Kernel Extensions and Device Support Programming Concepts*. If a Fortran program is not using Fortran asynchronous I/O statements, it will run regardless of the availability of AIX asynchronous I/O.

Linking

If there are no asynchronous I/O statements in an application, there is no change in the way you build an application. For example, for dynamic linking, you specify:

```
xlf95 -o t t.f
```

For static linking, you specify:

```
xlf95 -o t t.f -bnso -bnodelcsect -bI:/lib/syscalls.exp
```

If there are asynchronous I/O statements in an application, you need additional command-line options for static linking. For example:

```
xlf95 -o t t.f -lc -bnso -bnodelcsect \
      -bI:/lib/syscalls.exp -bI:/lib/aio.exp
```

Note that the additional options are **-lc** and **-bI:/lib/aio.exp**.

The following table summarizes the options that you need to bind applications in different situations:

Table 22. Table for Binding an Application Written Only in Fortran

	Fortran program using asynchronous I/O statements	
Type of Linking	**Yes**	**No**
Dynamic	xlf95 -o t t.f	xlf95 -o t t.f
Static	`xlf95 -o t t.f -bnso -bnodelcsect -bI:/lib/syscalls.exp -lc -bI:/lib/aio.exp`	`xlf95 -o t t.f -bnso -bnodelcsect -bI:/lib/syscalls.exp`

Table 23. Table for Binding an Application Written in Both Fortran and C, Where the C Routines Call the libc Asynchronous I/O Routines

	Fortran program using asynchronous I/O statements	
Type of Linking	**Yes**	**No**
Dynamic	xlf95 -o t t.f c.o -lc	xlf95 -o t t.f c.o -lc
Static	`xlf95 -o t t.f c.o -bnso -bnodelcsect -bI:/lib/syscalls.exp -lc -bI:/lib/aio.exp`	`xlf95 -o t t.f c.o -bnso -bnodelcsect -bI:/lib/syscalls.exp -lc -bI:/lib/aio.exp`
Note: *c.o* is an object file of routines written in C.		

You can bind an application that uses asynchronous I/O on a system with AIX asynchronous I/O disabled. However, you must run the resulting executable on a system with AIX asynchronous I/O enabled.

Error Handling

For an asynchronous data transfer, errors or end-of-file conditions might occur either during execution of the data transfer statement or during subsequent data transfer. If these conditions do not result in the termination of the program, you can detect these conditions via **ERR=**, **END=** and **IOSTAT=** specifiers in the data transfer or in the matching **WAIT** statement.

Execution of the program terminates if an error condition occurs during execution or during subsequent data transfer of an input/output statement that contains neither an **IOSTAT=** nor an **ERR=** specifier. In the case of a recoverable error, if the **IOSTAT=** and **ERR=** specifiers are not present, the program terminates if you set the **err_recovery** run-time option to **no**. If you set the **err_recovery** run-time option to **yes**, recovery action occurs, and the program continues.

If an asynchronous data transfer statement causes either of the following events, a matching **WAIT** statement cannot run, because the **ID=** value is not defined:
- A branch to the label that you specified by **ERR=** or **END=**
- The **IOSTAT=** specifier to be set to a non-zero value

XL Fortran Thread-Safe I/O Library

The XL Fortran thread-safe I/O library **libxlf90_r.a** provides support for parallel execution of Fortran I/O statements. For Fortran programs that contain I/O statements in a parallelized loop or that create multiple threads and execute I/O statements from different threads at the same time, you must use this library. In other words, to perform Fortran I/O in parallel, you must link applications with this library to get expected results.

However, note that on AIX operating system levels Version 4.3 and higher, a link is provided from the **libxlf90.a** library to the **libxlf90_r.a** library. You do not need to link with separate libraries depending on whether you are creating a threaded or a non-threaded application. XL Fortran determines at run time whether your application is threaded.

Synchronization of I/O Operations

During parallel execution, multiple threads might perform I/O operations on the same file at the same time. If they are not synchronized, the results of these I/O operations could be shuffled or merged or both, and the application might produce incorrect results or even terminate. The XL Fortran thread-safe I/O library synchronizes I/O operations for parallel applications. It performs the synchronization within the I/O library, and it is transparent to application programs. The purpose of the synchronization is to ensure the integrity and correctness of each individual I/O operation. However, the thread-safe I/O library does not have control over the order in which threads execute I/O statements. Therefore, the order of records read in or written out is not predictable under parallel I/O operations. Refer to "Parallel I/O Issues" for details.

External Files: For external files, the synchronization is performed on a per-unit basis. The XL Fortran thread-safe I/O library ensures that only one thread can access a particular logical unit to prevent several threads from interfering with each other. When a thread is performing an I/O operation on a unit, other threads attempting to perform I/O operations on the same unit must wait until the first thread finishes its operation. Therefore, the execution of I/O statements by multiple threads on the same unit is serialized. However, the thread-safe I/O library does not prevent threads from operating on different logical units in parallel. In other words, parallel access to different logical units is not necessarily serialized.

Functionality of I/O under Synchronization: The XL Fortran thread-safe I/O library sets its internal locks to synchronize access to logical units. This should not have any functional impact on the I/O operations performed by a Fortran program. Also, it will not impose any additional restrictions to the operability of Fortran I/O statements except for the use of I/O statements in a signal handler that is invoked asynchronously. Refer to "Use of I/O Statements in Signal Handlers" on page 343 for details.

The Fortran standard prohibits a function reference from appearing in an expression anywhere in an I/O statement if such a reference causes another I/O statement to run. This restriction still applies with the XL Fortran thread-safe I/O library.

Parallel I/O Issues

The order in which parallel threads perform I/O operations is not predictable. The XL Fortran thread-safe I/O library does not have control over the ordering. It will allow whichever thread that executes an I/O statement on a particular logical unit and obtains the lock on it first to proceed with the operation. Therefore, only use parallel I/O in cases where at least one of the following is true:

- Each thread performs I/O on a predetermined record in direct-access files.
- Each thread performs I/O on a different part of a stream-access file. Different I/O statements cannot use the same, or overlapping, areas of a file.

- The result of an application does not depend on the order in which records are written out or read in.
- Each thread performs I/O on a different file.

In these cases, results of the I/O operations are independent of the order in which threads execute. However, you might not get the performance improvements that you expect, since the I/O library serializes parallel access to the same logical unit from multiple threads. Examples of these cases are as follows:

- Each thread performs I/O on a pre-determined record in a direct-access file:

```
do i = 1, 10
  write(4, '(i4)', rec = i) a(i)
enddo
```

- Each thread performs I/O on a different part of a stream-access file. Different I/O statements cannot use the same, or overlapping, areas of a file.

```
do i = 1, 9
  write(4, '(i4)', pos = 1 + 5 * (i - 1)) a(i)
  ! We use 5 above because i4 takes 4 file storage
  ! units + 1 file storage unit for the record marker.
enddo
```

- In the case that each thread operates on a different file, since threads share the status of the logical units connected to the files, the thread still needs to obtain the lock on the logical unit for either retrieving or updating the status of the logical unit. However, the thread-safe I/O library allows threads to perform the data transfer between the logical unit and the I/O list item in parallel. If an application contains a large number of small I/O requests in a parallel region, you might not get the expected performance because of the lock contention. Consider the following example:

```
      program example

      use omp_lib

      integer, parameter :: num_of_threads = 4, max = 5000000
      character*10 file_name
      integer i, file_unit, thread_id
      integer, dimension(max, 2 * num_of_threads) :: aa

      call omp_set_num_threads(num_of_threads)

!$omp parallel private(file_name, thread_id, file_unit, i) shared(aa)

      thread_id = omp_get_thread_num()
      file_name = 'file_'
      file_name(6:6) = char(ichar('0') + thread_id)
      file_unit = 10 + thread_id

      open(file_unit, file = file_name, status = 'old', action = 'read')

      do i = 1, max
         read(file_unit, *) aa(i, thread_id * 2 + 1), aa(i, thread_id * 2 + 2)
      end do

      close(file_unit)

!$omp end parallel
      end
```

The I/O library synchronizes retrieving and updating the status of the logical units while performing data transfer in parallel. In order to maximize the parallelism the I/O library provides, it is recommanded to increase the size of data transfer per I/O request. The do loop, therefore, should be rewritten as follows:

```
read(file_unit, *) a(:, thread_id * 2 + 1 : thread_id * 2 + 2)

do i = 1, max
   ! Do something for each element of array 'aa'.
end do
```

- The result does not depend on the order in which records are written out or read in:

  ```
  real a(100)
  do i = 1, 10
    read(4) a(i)
  enddo
  call qsort_(a)
  ```
- Each thread performs I/O on a different logical unit of direct access, sequential access, or stream access:

  ```
  do i = 11, 20
    write(i, '(i4)') a(i - 10)
  enddo
  ```

For multiple threads to write to or read from the same sequential-access file, or to write to or read from the same stream-access file without using the **POS=** specifier, the order of records written out or read in depends on the order in which the threads execute the I/O statement on them. This order, as stated previously, is not predictable. Therefore, the result of an application could be incorrect if it assumes records are sequentially related and cannot be arbitrarily written out or read in. For example, if the following loop is parallelized, the numbers printed out will no longer be in the sequential order from 1 to 500 as the result of a serial execution:

```
do i = 1, 500
  print *, i
enddo
```

Applications that depend on numbers being strictly in the specified order will not work correctly.

The XL Fortran run-time option **multconn=yes** allows connection of the same file to more than one logical unit simultaneously. Since such connections can only be made for reading (**ACCESS='READ'**), access from multiple threads to logical units that are connected to the same file will produce predictable results.

Use of I/O Statements in Signal Handlers

There are basically two kinds of signals in the POSIX signal model: *synchronously* and *asynchronously* generated signals. Signals caused by the execution of some code of a thread, such as a reference to an unmapped, protected, or bad memory (**SIGSEGV** or **SIGBUS**), floating-point exception (**SIGFPE**), execution of a trap instruction (**SIGTRAP**), or execution of illegal instructions (**SIGILL**) are said to be synchronously generated. Signals may also be generated by events outside the process: for example, **SIGINT**, **SIGHUP**, **SIGQUIT**, **SIGIO**, and so on. Such events are referred to as interrupts. Signals that are generated by interrupts are said to be asynchronously generated.

The XL Fortran thread-safe I/O library is asynchronous signal unsafe. This means that the XL Fortran I/O statements cannot be used in a signal handler that is entered because of an asynchronously generated signal. The behavior of the system is undefined when an XL Fortran I/O statement is called from a signal handler that interrupts an I/O statement. However, it is safe to use I/O statements in signal handlers for synchronous signals.

Sometimes an application can guarantee that a signal handler is not entered asynchronously. For example, an application might mask signals except when it runs certain known sections of code. In such situations, the signal will not interrupt any I/O statements and other asynchronous signal unsafe functions. Therefore, you can still use Fortran I/O statements in an asynchronous signal handler.

A much easier and safer way to handle asynchronous signals is to block signals in all threads and to explicitly wait (using **sigwait()**) for them in one or more separate threads. The advantage of this approach is that the **handler** thread can use Fortran I/O statements as well as other asynchronous signal unsafe routines.

Asynchronous Thread Cancellation

When a thread enables asynchronous thread cancellability, any cancellation request is acted upon immediately. The XL Fortran thread-safe I/O library is not asynchronous thread cancellation safe. The behavior of the system is undefined if a thread is cancelled asynchronously while it is in the XL Fortran thread-safe I/O library.

Interlanguage Calls

This section gives details about performing interlanguage calls from your Fortran program: that is, calling routines that were written in a language other than Fortran. It assumes that you are familiar with the syntax of all applicable languages.

Conventions for XL Fortran External Names

To assist you in writing mixed-language programs, XL Fortran follows a consistent set of rules when translating the name of a global entity into an external name that the linker can resolve:

- Both the underscore (_) and the dollar sign ($) are valid characters anywhere in names.

  Because names that begin with an underscore are reserved for the names of library routines, do not use an underscore as the first character of a Fortran external name.

  To avoid conflicts between Fortran and non-Fortran function names, you can compile the Fortran program with the **-qextname** option. This option adds an underscore to the end of the Fortran names. Then use an underscore as the last character of any non-Fortran procedures that you want to call from Fortran.
- Names can be up to 250 characters long.
- Program and symbolic names are interpreted as all lowercase by default. If you are writing new non-Fortran code, use all-lowercase procedure names to simplify calling the procedures from Fortran.

  You can use the **-U** option or the **@PROCESS MIXED** directive if you want the names to use both uppercase and lowercase:

```
@process mixed
       external C_Func    ! With MIXED, we can call C_Func, not just c_func.
       integer aBc, ABC   ! With MIXED, these are different variables.
       common /xYz/ aBc   ! The same applies to the common block names.
       common /XYZ/ ABC   ! xYz and XYZ are external names that are
                          ! visible during linking.
       end
```

- Names for module procedures are formed by concatenating __ (two underscores), the module name, `_IMOD_` (for intrinsc modules) or `_NMOD_` (for non-intrinsic modules), and the name of the module procedure. For example, module procedure `MYPROC` in module `MYMOD` has the external name `__mymod_NMOD_myproc`.

- The XL compilers generate code that uses `main` as an external entry point name. You can only use `main` as an external name in these contexts:
  - A Fortran program or local-variable name. (This restriction means that you cannot use `main` for the name of an external function, external subroutine, block data program unit, or common block. References to such an object use the compiler-generated **main** instead of your own.)
  - The name of the top-level main function in a C program.
  - The name of a Pascal program unit.
- Some other potential naming conflicts may occur when linking a program. For instructions on avoiding them, see "Linking New Objects with Existing Ones" on page 45 and "Avoiding Naming Conflicts during Linking" on page 47.

If you are porting your application from another system and your application does encounter naming conflicts like these, you may need to use the "-qextname Option" on page 158. Or you can use the **-brename** linker option to rename the symbol if there are not too many names to change:

```
xlf90 -brename:old_name,new_name interlanguage_calls.f
```

Mixed-Language Input and Output

To improve performance, the XL Fortran run-time library has its own buffers and its own handling of these buffers. This means that mixed-language programs cannot freely mix I/O operations on the same file from the different languages. To maintain data integrity in such cases:

- If the file position is not important, open and explicitly close the file within the Fortran part of the program before performing any I/O operations on that file from subprograms written in another language.
- To open a file in Fortran and manipulate the open file from another language, call the **flush_** procedure to save any buffer for that file, and then use the **getfd** procedure to find the corresponding file descriptor and pass it to the non-Fortran subprogram. As an alternative to calling the **flush_** procedure, you can use the **buffering** run-time option to disable the buffering for I/O operations. When you specify **buffering=disable_preconn**, XL Fortran disables the buffering for preconnected units. When you specify **buffering=disable_all**, XL Fortran disables the buffering for all logical units.

  Note: After you call **flush_** to flush the buffer for a file, do not do anything to the file from the Fortran part of the program except to close it when the non-Fortran processing is finished.
- If any XL Fortran subprograms containing **WRITE** statements are called from a non-Fortran main program, explicitly **CLOSE** the data file, or use the **flush_** subroutine in the XL Fortran subprograms to ensure that the buffers are flushed. Alternatively, you can use the **buffering** run-time option to disable buffering for I/O operations.

Related Information: For more information on the **flush_** and **getfd** procedures, see the *Service and Utility Procedures* section in the *XL Fortran Enterprise Edition for AIX Language Reference*. For more information on the **buffering** run-time option, see "Setting Run-Time Options" on page 51.

Mixing Fortran and C++

Most of the information in this section applies to Fortran, C, and Pascal — languages with similar data types and naming schemes. However, to mix Fortran and C++ in the same program, you must add an extra level of indirection and pass the interlanguage calls through C wrapper functions.

Because the C++ compiler mangles the names of some C++ objects, you must use the **xlC** command to link the final program and include **-L** and **-l** options for the XL Fortran library directories and libraries as shown in "Linking 32–Bit Non-SMP Object Files Using the ld Command" on page 44.

```
program main

integer idim,idim1

idim = 35
idim1= 45

write(6,*) 'Inside Fortran calling first C function'
call cfun(idim)
write(6,*) 'Inside Fortran calling second C function'
call cfun1(idim1)
write(6,*) 'Exiting the Fortran program'
end
```

Figure 1. Main Fortran Program That Calls C++ (main1.f)

```
 #include <stdio.h>
 #include "cplus.h"

extern "C" void cfun(int *idim);
extern "C" void cfun1(int *idim1);

 void cfun(int *idim){
   printf("%%%Inside C function before creating C++ Object\n");
   int i = *idim;
   junk<int>* jj= new junk<int>(10,30);
   jj->store(idim);
   jj->print();
   printf("%%%Inside C function after creating C++ Object\n");
   delete jj;
   return;
 }

 void cfun1(int *idim1) {
   printf("%%%Inside C function cfun1 before creating C++ Object\n");
   int i = *idim1;
   temp<double> *tmp = new temp<double>(40, 50.54);
   tmp->print();
   printf("%%%Inside C function after creating C++ temp object\n");
   delete tmp;
   return;
 }
```

Figure 2. C Wrapper Functions for Calling C++ (cfun.C)

```
#include <iostream.h>

template<class T> class junk {

  private:
   int inter;
   T   templ_mem;
   T   stor_val;

  public:
   junk(int i,T j): inter(i),templ_mem(j)
                   {cout <<"***Inside C++ constructor" << endl;}

    ~junk()         {cout <<"***Inside C++ Destructor"  << endl;}

   void store(T *val){ stor_val = *val;}

   void print(void) {cout << inter << "\t" << templ_mem ;
                    cout <<"\t" << stor_val << endl; }};

template<class T> class temp {

  private:
    int internal;
    T temp_var;

  public:
    temp(int i, T j): internal(i),temp_var(j)
                    {cout <<"***Inside C++ temp Constructor" <<endl;}

     ~temp()         {cout <<"***Inside C++ temp destructor"  <<endl;}

    void print(void) {cout << internal << "\t" << temp_var << endl;}};
```

Figure 3. C++ Code Called from Fortran (cplus.h)

Compiling this program, linking it with the **xlC** command, and running it produces this output:

```
 Inside Fortran calling first C function
%Inside C function before creating C++ Object
***Inside C++ constructor
10     30     35
%Inside C function after creating C++ Object
***Inside C++ Destructor
 Inside Fortran calling second C function
%Inside C function cfun1 before creating C++ Object
***Inside C++ temp Constructor
40     50.54
%Inside C function after creating C++ temp object
***Inside C++ temp destructor
 Exiting the Fortran program
```

Making Calls to C Functions Work

When you pass an argument to a subprogram call, the usual Fortran convention is to pass the address of the argument. Many C functions expect arguments to be passed as values, however, not as addresses. For these arguments, specify them as **%VAL**(*argument*) in the call to C, as follows:

```
 MEMBLK = MALLOC(1024)    ! Wrong, passes the address of the constant
 MEMBLK = MALLOC(N)       ! Wrong, passes the address of the variable

 MEMBLK = MALLOC(%VAL(1024)) ! Right, passes the value 1024
 MEMBLK = MALLOC(%VAL(N))    ! Right, passes the value of the variable
```

See “Passing Arguments By Reference or By Value” on page 353 and *%VAL and %REF* in the *XL Fortran Enterprise Edition for AIX Language Reference* for more details.

Passing Data From One Language to Another

The following table shows the data types available in the XL Fortran, Pascal, and C languages. This section shows how Fortran arguments can be passed by reference to C programs. To use the Fortran 2003 Draft Standard interoperability features, such as the **BIND(C)** attribute and **ISO_C_BINDING** module support, see the *XL Fortran Enterprise Edition for AIX Language Reference*.

Passing Arguments between Languages

Table 24. Corresponding Data Types in Fortran, C, and Pascal. When calling Fortran, the C and Pascal routines must pass arguments as pointers to the types listed in this table.

XL Fortran Data Types	IBM C Data Types	XL Pascal Data Types
INTEGER(1), BYTE	signed char	PACKED -128..127
INTEGER(2)	signed short	PACKED -32768..32767
INTEGER(4)	signed int	INTEGER
INTEGER(8)	signed long long (see note 1)	—
REAL, REAL(4)	float	SHORTREAL
REAL(8), DOUBLE PRECISION	double	REAL
REAL(16)	long double (see note 2)	—
COMPLEX, COMPLEX(4)	structure of two floats	record of two SHORTREALs
COMPLEX(8), DOUBLE COMPLEX	structure of two doubles	record of two REALs
COMPLEX(16)	structure of two long doubles	—
LOGICAL(1)	unsigned char	PACKED 0..255
LOGICAL(2)	unsigned short	PACKED 0..65535
LOGICAL(4)	unsigned int	—
LOGICAL(8)	unsigned long long (see note 1)	—
CHARACTER	char	CHAR
CHARACTER(n)	char[n]	PACKED ARRAY[1..n] OF CHAR
Integer POINTER	void *	POINTER, or typed pointer such as @INTEGER (see note 3)
Array	array	ARRAY
Sequence-derived type	structure (with C -qalign=packed option)	PACKED RECORD

Notes:
1. Requires the option -qlanglvl=extended in XL C or -qlonglong in C for AIX and C Set ++ for AIX. These are default options for some compilation commands but not for others.
2. Requires C compiler -qlongdbl option.
3. Requires XL Pascal -qptr4 option.

Notes:

1. In interlanguage communication, it is often necessary to use the **%VAL** and **%REF** built-in functions that are defined in "Passing Arguments By Reference or By Value" on page 353.
2. C programs automatically convert float values to double and short integer values to integer when calling an unprototyped C function. Because XL Fortran does not perform a conversion on **REAL(4)** quantities passed by value, you should not pass **REAL(4)** and **INTEGER(2)** values as arguments to C functions that you have not declared with function prototypes.
3. The Fortran-derived type, the Pascal RECORD, and the C structure must match in the number, data type, and length of subobjects to be compatible data types.

Related Information: One or more sample programs under the directory **/usr/lpp/xlf/samples** illustrate how to call from Fortran to C.

To use the Fortran 2003 Draft Standard interoperability features provided by XL Fortran, see the *Language Interoperability Features* section in the *XL Fortran Enterprise Edition for AIX Language Reference*.

Passing Global Variables between Languages

To access a C data structure from within a Fortran program or to access a common block from within a C program, follow these steps:

1. Create a named common block that provides a one-to-one mapping of the C structure members. If you have an unnamed common block, change it to a named one. Name the common block with the name of the C structure.
2. Declare the C structure as a global variable by putting its declaration outside any function or inside a function with the **extern** qualifier.
3. Compile the C source file with **-qalign=packed**.

```
program cstruct                     struct mystuff {
real(8) a,d                                double a;
integer b,c                                int b,c;
.                                          double d;
.                                   };
common /mystuff/ a,b,c,d
.                                   main() {
.
end                                 }
```

If you do not have a specific need for a named common block, you can create a sequence-derived type with the same one-to-one mapping as a C structure and pass it as an argument to a C function. You must compile the C source file with **-qalign=packed**.

Common blocks that are declared **THREADLOCAL** are thread-specific data areas that are dynamically allocated by compiler-generated code. A static block is still reserved for a **THREADLOCAL** common block, but the compiler and the compiler's run-time environment use it for control information. If you need to share **THREADLOCAL** common blocks between Fortran and C procedures, your C source must be aware of the implementation of the **THREADLOCAL** common block. For more information, see **THREADLOCAL** common blocks, the *Directives* section in the *XL Fortran Enterprise Edition for AIX Language Reference*, and Appendix A, "Sample Fortran Programs," on page 405.

Common blocks that are declared **THREADPRIVATE** can be accessed using a C global variable that is declared as **THREADPRIVATE**, using C for AIX 4.5 or later.

Passing Character Types between Languages

One difficult aspect of interlanguage calls is passing character strings between languages. The difficulty is due to the following underlying differences in the way that different languages represent such entities:

- The only character type in Fortran is **CHARACTER**, which is stored as a set of contiguous bytes, one character per byte. The length is not stored as part of the entity. Instead, it is passed by value as an extra argument at the end of the declared argument list when the entity is passed as an argument.
- Character strings in C are stored as arrays of the type **char**. A null character indicates the end of the string.

  Note: To have the compiler automatically add the null character to certain character arguments, you can use the "-qnullterm Option" on page 206.
- Pascal's character-variable data types are **STRING**, **PACKED ARRAY OF CHAR**, **GSTRING**, and **PACKED ARRAY OF GCHAR**. The **STRING** data type has a two-byte string length that is usually aligned on a half-word boundary followed by a set of contiguous bytes, one character per byte. The dynamic length of the string can be determined using the predefined Pascal function **LENGTH**. Packed arrays of **CHAR**, like Fortran's **CHARACTER** type, are stored as a set of contiguous bytes, one character per byte.

If you are writing both parts of the mixed-language program, you can make the C routines deal with the extra Fortran length argument, or you can suppress this extra argument by passing the string using the **%REF** function. If you use **%REF**, which you typically would for pre-existing C routines, you need to indicate where the string ends by concatenating a null character to the end of each character string that is passed to a C routine:

```
! Initialize a character string to pass to C.
              character*6 message1 /'Hello\0'/
! Initialize a character string as usual, and append the null later.
              character*5 message2 /'world'/

! Pass both strings to a C function that takes 2 (char *) arguments.
              call cfunc(%ref(message1), %ref(message2 // '\0'))
              end
```

For compatibility with C language usage, you can encode the following escape sequences in XL Fortran character strings:

Table 25. Escape Sequences for Character Strings

Escape	Meaning
\b	Backspace
\f	Form feed
\n	New-line
\t	Tab
\0	Null
\'	Apostrophe (does not terminate a string)
\"	Double quotation mark (does not terminate a string)
\ \	Backslash

Table 25. Escape Sequences for Character Strings (continued)

Escape	Meaning
x	*x*, where *x* is any other character (the backslash is ignored)

If you do not want the backslash interpreted as an escape character within strings, you can compile with the **-qnoescape** option.

Passing Arrays between Languages

Fortran stores array elements in ascending storage units in column-major order. C and Pascal store array elements in row-major order. Fortran and Pascal array indexes start at 1, while C array indexes start at 0.

The following example shows how a two-dimensional array that is declared by A(3,2) is stored in Fortran, C, and Pascal:

Table 26. Corresponding Array Layouts for Fortran, C, and Pascal. The Fortran array reference `A(X,Y,Z)` can be expressed in C as `a[Z-1][Y-1][X-1]` and in Pascal as `A[Z,Y,X]`. Keep in mind that although C passes individual scalar array elements by value, it passes arrays by reference.

	Fortran Element Name	C Element Name	Pascal Element Name
Lowest storage unit	A(1,1)	A[0][0]	A[1,1]
	A(2,1)	A[0][1]	A[1,2]
	A(3,1)	A[1][0]	A[2,1]
	A(1,2)	A[1][1]	A[2,2]
	A(2,2)	A[2][0]	A[3,1]
Highest storage unit	A(3,2)	A[2][1]	A[3,2]

To pass all or part of a Fortran array to another language, you can use Fortran 90/Fortran 95 array notation:

```
REAL, DIMENSION(4,8) :: A, B(10)

! Pass an entire 4 x 8 array.
CALL CFUNC( A )
! Pass only the upper-left quadrant of the array.
CALL CFUNC( A(1:2,1:4) )
! Pass an array consisting of every third element of A.
CALL CFUNC( A(1:4:3,1:8) )
! Pass a 1-dimensional array consisting of elements 1, 2, and 4 of B.
CALL CFUNC( B( (/1,2,4/) ) )
```

Where necessary, the Fortran program constructs a temporary array and copies all the elements into contiguous storage. In all cases, the C routine needs to account for the column-major layout of the array.

Any array section or noncontiguous array is passed as the address of a contiguous temporary unless an explicit interface exists where the corresponding dummy argument is declared as an assumed-shape array or a pointer. To avoid the creation of array descriptors (which are not supported for interlanguage calls) when calling non-Fortran procedures with array arguments, either do not give the non-Fortran procedures any explicit interface, or do not declare the corresponding dummy arguments as assumed-shape or pointers in the interface:

```
! This explicit interface must be changed before the C function
! can be called.
INTERFACE
  FUNCTION CFUNC (ARRAY, PTR1, PTR2)
    INTEGER, DIMENSION (:) :: ARRAY          ! Change this : to *.
    INTEGER, POINTER, DIMENSION (:) :: PTR1  ! Change this : to *
                                             ! and remove the POINTER
                                             ! attribute.
    REAL, POINTER :: PTR2                    ! Remove this POINTER
                                             ! attribute or change to TARGET.
  END FUNCTION
END INTERFACE
```

Passing Pointers between Languages

Integer **POINTER**s always represent the address of the pointee object and must be passed by value:

```
CALL CFUNC(%VAL(INTPTR))
```

Note that the FORTRAN 77 **POINTER** extension from XL Fortran Version 2 is now referred to as "integer **POINTER**" to distinguish it from the Fortran 90 meaning of **POINTER**.

Fortran 90 **POINTER**s can also be passed back and forth between languages but only if there is no explicit interface for the called procedure or if the argument in the explicit interface does not have a **POINTER** attribute or assumed-shape declarator. You can remove any **POINTER** attribute or change it to **TARGET** and can change any deferred-shape array declarator to be explicit-shape or assumed-size.

Because of XL Fortran's call-by-reference conventions, you must pass even scalar values from another language as the address of the value, rather than the value itself. For example, a C function passing an integer value `x` to Fortran must pass `&x`. Also, a C function passing a pointer value `p` to Fortran so that Fortran can use it as an integer **POINTER** must declare it as `void **p`. A C array is an exception: you can pass it to Fortran without the `&` operator.

Passing Arguments By Reference or By Value

To call subprograms written in languages other than Fortran (for example, user-written C programs, or operating system routines), the actual arguments may need to be passed by a method different from the default method used by Fortran. C routines, including those in system libraries such as **libc.a**, require you to pass arguments by value instead of by reference. (Although C passes individual scalar array elements by value, it passes arrays by reference.)

You can change the default passing method by using the **%VAL** and **%REF** built-in functions in the argument list of a **CALL** statement or function reference. You cannot use them in the argument lists of Fortran procedure references or with alternate return specifiers.

%REF Passes an argument by reference (that is, the called subprogram receives the address of the argument). It is the same as the default calling method for Fortran except that it also suppresses the extra length argument for character strings.

%VAL Passes an argument by value (that is, the called subprogram receives an argument that has the same value as the actual argument, but any change to this argument does not affect the actual argument).

You can use this built-in function with actual arguments that are **CHARACTER(1)**, **BYTE**, logical, integer, real, or complex expressions or that are sequence-derived type. Objects of derived type cannot contain pointers, arrays, or character structure components whose lengths are greater than one byte.

You cannot use **%VAL** with actual arguments that are array entities, procedure names, or character expressions of length greater than one byte.

%VAL causes XL Fortran to pass the actual argument as 32-bit or 64-bit intermediate values.

In 32-bit Mode

If the actual argument is one of the following:

- An integer or a logical that is shorter than 32 bits, it is sign-extended to a 32-bit value.
- An integer or a logical that is longer than 32 bits, it is passed as two 32-bit intermediate values.
- Of type real or complex, it is passed as multiple 64-bit intermediate values.
- Of sequence-derived type, it is passed as multiple 32-bit intermediate values.

Byte-named constants and variables are passed as if they were **INTEGER(1)**. If the actual argument is a **CHARACTER(1)**, the compiler pads it on the left with zeros to a 32-bit value, regardless of whether you specified the **-qctyplss** compiler option.

In 64-bit Mode

If the actual argument is one of the following:

- An integer or a logical that is shorter than 64 bits, it is sign-extended to a 64-bit value.
- Of type real or complex, it is passed as multiple 64-bit intermediate values.
- Of sequence-derived type, it is passed as multiple 64-bit intermediate values.

Byte-named constants and variables are passed as if they were **INTEGER(1)**. If the actual argument is a **CHARACTER(1)**, the compiler pads it on the left with zeros to a 64-bit value, regardless of whether you specified the **-qctyplss** compiler option.

If you specified the **-qautodbl** compiler option, any padded storage space is not passed except for objects of derived type.

```
EXTERNAL FUNC
COMPLEX XVAR
IVARB=6

CALL RIGHT2(%REF(FUNC))       ! procedure name passed by reference
CALL RIGHT3(%VAL(XVAR))       ! complex argument passed by value
CALL TPROG(%VAL(IVARB))       ! integer argument passed by value
END
```

Explicit Interface for %VAL and %REF

You can specify an explicit interface for non-Fortran procedures to avoid coding calls to **%VAL** and **%REF** in each argument list, as follows:

```
INTERFACE
    FUNCTION C_FUNC(%VAL(A),%VAL(B)) ! Now you can code "c_func(a,b)"
        INTEGER A,B                  ! instead of
    END FUNCTION C_FUNC              ! "c_func(%val(a),%val(b))".
END INTERFACE
```

Returning Values from Fortran Functions

XL Fortran does not support calling certain types of Fortran functions from non-Fortran procedures. If a Fortran function returns a pointer, array, or character of nonconstant length, do not call it from outside Fortran.

You can call such a function indirectly:

```
SUBROUTINE MAT2(A,B,C)    ! You can call this subroutine from C, and the
                          ! result is stored in C.
INTEGER, DIMENSION(10,10) :: A,B,C
C = ARRAY_FUNC(A,B)       ! But you could not call ARRAY_FUNC directly.
END
```

Arguments with the OPTIONAL Attribute

When you pass an optional argument by reference, the address in the argument list is zero if the argument is not present.

When you pass an optional argument by value, the value is zero if the argument is not present. The compiler uses an extra register argument to differentiate that value from a regular zero value. If the register has the value 1, the optional argument is present; if it has the value 0, the optional argument is not present.

Related Information: See “Order of Arguments in Argument List” on page 365.

Arguments with the INTENT Attribute

Currently, declaring arguments with the **INTENT** attribute does not change the linkage convention for a procedure. However, because this part of the convention is subject to change in the future, we recommend not calling from non-Fortran procedures into Fortran procedures that have **INTENT(IN)** arguments.

Type Encoding and Checking

Run-time errors are hard to find, and many of them are caused by mismatched procedure interfaces or conflicting data definitions. Therefore, it is a good idea to find as many of these problems as possible at compile or link time. To store type information in the object file so that the linker can detect mismatches, use the **-qextchk** compiler option.

Assembler-Level Subroutine Linkage Conventions

The subroutine linkage convention specifies the machine state at subroutine entry and exit, allowing routines that are compiled separately in the same or different languages to be linked. The information on subroutine linkage and system calls in the *AIX Commands Reference* is the base reference on this topic. You should consult it for full details. This section summarizes the information needed to write mixed-language Fortran and assembler programs or to debug at the assembler level, where you need to be concerned with these kinds of low-level details.

The system linkage convention passes arguments in registers, taking full advantage of the large number of floating-point registers (FPRs) and general-purpose registers (GPRs) and minimizing the saving and restoring of registers on subroutine entry and exit. The linkage convention allows for argument passing and return values to be in FPRs, GPRs, or both.

The following table lists floating-point registers and their functions. The floating-point registers are double precision (64 bits).

Table 27. Floating-Point Register Usage across Calls

Register	Preserved Across Calls	Use
0	no	
1	no	FP parameter 1, function return 1.
2	no	FP parameter 2, function return 2.
⋮	⋮	⋮
13	no	FP parameter 13, function return 13.
14-31	yes	

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

Table 28. General-Purpose Register Usage across Calls

Register	Preserved Across Calls	Use
0	no	
1	yes	Stack pointer.
2	yes	TOC pointer.
3	no	1st word of arg list; return value 1.
4	no	2nd word of arg list; return value 2.
⋮	⋮	⋮
10	no	8th word of arg list; return value 8.
11	no	DSA pointer to internal procedure (Env).
12	no	
13-31	yes	
If a register is not designated as preserved, its contents may be changed during the call, and the caller is responsible for saving any registers whose values are needed later. Conversely, if a register is supposed to be preserved, the callee is responsible for preserving its contents across the call, and the caller does not need any special action.		

The following table lists special-purpose register conventions.

Table 29. Special-Purpose Register Usage across Calls

Register	Preserved Across Calls
Condition register	
Bits 0-7 (CR0,CR1)	no
Bits 8-22 (CR2,CR3,CR4)	yes
Bits 23-31 (CR5,CR6,CR7)	no
Link register	no

Table 29. Special-Purpose Register Usage across Calls (continued)

Register	Preserved Across Calls
Count register	no
MQ register	no
XER register	no
FPSCR register	no

The Stack

The stack is a portion of storage that is used to hold local storage, register save areas, parameter lists, and call-chain data. The stack grows from higher addresses to lower addresses. A stack pointer register (register 1) is used to mark the current “top” of the stack.

A stack frame is the portion of the stack that is used by a single procedure. The input parameters are considered part of the current stack frame. In a sense, each output argument belongs to both the caller’s and the callee’s stack frames. In either case, the stack frame size is best defined as the difference between the caller’s stack pointer and the callee’s.

The following diagrams show the storage maps of typical stack frames for 32-bit and 64-bit environments.

In these diagrams, the current routine has acquired a stack frame that allows it to call other functions. If the routine does not make any calls and there are no local variables or temporaries, the function need not allocate a stack frame. It can still use the register save area at the top of the caller’s stack frame, if needed.

The stack frame is double-word aligned. The FPR save area and the parameter area (P1, P2, ..., Pn) are double-word aligned. Other areas require word alignment only.

Run-time Stack for 32-bit Environment

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

Run-time Stack for 64-bit Environment

```
  Low                  |                    |  Stack grows at
  Addresses            |                    |  this end.
                       |--------------------|
Callee's stack  --> 0  |     Back chain     |
pointer             8  |      Saved CR      |
                   16  |      Saved LR      |
                24-32  |      Reserved      | <--- LINK AREA
                   40  |      Saved TOC     |      (callee)
                       |--------------------|
 Space for P1-P8       |         P1         |  OUTPUT ARGUMENT AREA
 is always reserved    |         ...        | <---(Used by callee
                       |         Pn         |      to construct
                       |--------------------|      argument list)
                       |      Callee's      |
                       |       stack        | <--- LOCAL STACK AREA
                       |        area        |
                       |--------------------|
                       |                    | (Possible word wasted
                       |--------------------|    for alignment.)
 -8*nfprs-8*ngprs -->  |    Save area for   | Rfirst = R13 for full
   save                |    caller's GPR    |          save
                       | max 19 doublewords | R31
                       |--------------------|
         -8*nfprs -->  |    Save area for   | Ffirst = F14 for a
                       |    caller's FPR    |          full save
                       |    max 18 dblwds   | F31
                       |--------------------|
Caller's stack  --> 0  |     Back chain     |
 pointer            8  |      Saved CR      |
                   16  |      Saved LR      |
                24-32  |      Reserved      | <--- LINK AREA
                   40  |      Saved TOC     |      (caller)
                       |--------------------|
Space for P1-P8    48  |         P1         |  INPUT PARAMETER AREA
is always reserved     |         ...        | <---(Callee's input
                       |         Pn         |   parameters found
                       |--------------------|    here. Is also
                       |      Caller's      |   caller's arg area.)
                       |       stack        |
  High                 |        area        |
  Addresses            |                    |
```

The Link Area

In a 32-bit environment, the link area consists of six words at offset zero from the caller's stack pointer on entry to a procedure. The first word contains the caller's back chain (stack pointer). The second word is the location where the callee saves the Condition Register (CR) if it is needed. The third word is the location where the callee's prolog code saves the Link Register if it is needed. The fourth word is reserved for C **SETJMP** and **LONGJMP** processing, and the fifth word is reserved for future use. The last word (word 6) is reserved for use by the global linkage routines that are used when calling routines in other object modules (for example, in shared libraries).

In a 64-bit environment, this area consists of six doublewords at offset zero from the caller's stack pointer on entry to a procedure. The first doubleword contains the caller's back chain (stack pointer). The second doubleword is the location where the callee saves the Condition Register (CR) if it is needed. The third doubleword is the location where the callee's prolog code saves the Link Register if it is needed. The fourth doubleword is reserved for C **SETJMP** and **LONGJMP** processing, and the fifth doubleword is reserved for future use. The last doubleword (doubleword 6) is reserved for use by the global linkage routines that are used when calling routines in other object modules (for example, in shared libraries).

The Input Parameter Area

In a 32-bit environment, the input parameter area is a contiguous piece of storage reserved by the calling program to represent the register image of the input parameters of the callee. The input parameter area is double-word aligned and is located on the stack directly following the caller's link area. This area is at least 8 words in size. If more than 8 words of parameters are expected, they are stored as register images that start at positive offset 56 from the incoming stack pointer.

The first 8 words only appear in registers at the call point, never in the stack. Remaining words are always in the stack, and they can also be in registers.

In a 64-bit environment, the input parameter area is a contiguous piece of storage reserved by the calling program to represent the register image of the input parameters of the callee. The input parameter area is double-word aligned and is located on the stack directly following the caller's link area. This area is at least 8 doublewords in size. If more than 8 doublewords of parameters are expected, they are stored as register images that start at positive offset 112 from the incoming stack pointer.

The first 8 doublewords only appear in registers at the call point, never in the stack. Remaining words are always in the stack, and they can also be in registers.

The Register Save Area

The register save area is double-word aligned. It provides the space that is needed to save all nonvolatile FPRs and GPRs used by the callee program. The FPRs are saved next to the link area. The GPRs are saved above the FPRs (in lower addresses). The called function may save the registers here even if it does not need to allocate a new stack frame. The system-defined stack floor includes the maximum possible save area:

```
32-bit platforms:  18*8 for FPRs + 19*4 for GPRs
64-bit platforms:  18*8 for FPRs + 19*8 for GPRs
```

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

A callee needs only to save the nonvolatile registers that it actually uses. It always saves register 31 in the highest
- addressed word (in a 32-bit environment)
- addressed doubleword (in a 64-bit environment)

The Local Stack Area

The local stack area is the space that is allocated by the callee procedure for local variables and temporaries.

The Output Parameter Area

The output parameter area (P1...Pn) must be large enough to hold the largest parameter list of all procedures that the procedure that owns this stack frame calls.

In a 32-bit environment, this area is at least 8 words long, regardless of the length or existence of any argument list. If more than 8 words are being passed, an extension list is constructed beginning at offset 56 from the current stack pointer.

The first 8 words only appear in registers at the call point, never in the stack. Remaining words are always in the stack, and they can also be in registers.

In a 64-bit environment, this area is at least 8 doublewords long, regardless of the length or existence of any argument list. If more than 8 doublewords are being passed, an extension list is constructed, which begins at offset 112 from the current stack pointer.

The first 8 doublewords only appear in registers at the call point, never in the stack. Remaining doublewords are always in the stack, and they can also be in registers.

Linkage Convention for Argument Passing

The system linkage convention takes advantage of the large number of registers available. The linkage convention passes arguments in both GPRs and FPRs. Two fixed lists, R3-R10 and FP1-FP13, specify the GPRs and FPRs available for argument passing.

When there are more argument words than available argument GPRs and FPRs, the remaining words are passed in storage on the stack. The values in storage are the same as if they were in registers.

The size of the parameter area is sufficient to contain all the arguments passed on any call statement from a procedure that is associated with the stack frame. Although not all the arguments for a particular call actually appear in storage, it is convenient to consider them as forming a list in this area, each one occupying one or more words.

For call by reference (as is the default for Fortran), the address of the argument is passed in a register. The following information refers to call by value, as in C or as in Fortran when **%VAL** is used. For purposes of their appearance in the list, arguments are classified as floating-point values or non-floating-point values:

In a 32-bit Environment

- Each **INTEGER(8)** and **LOGICAL(8)** argument requires two words.
- Any other non-floating-point scalar argument of intrinsic type requires one word and appears in that word exactly as it would appear in a GPR. It is right-justified, if language semantics specify, and is word aligned.
- Each single-precision (**REAL(4)**) value occupies one word. Each double-precision (**REAL(8)**) value occupies two successive words in the list. Each extended-precision (**REAL(16)**) value occupies four successive words in the list.
- A **COMPLEX** value occupies twice as many words as a **REAL** value with the same kind type parameter.
- In Fortran and C, structure values appear in successive words as they would anywhere in storage, satisfying all appropriate alignment requirements. Structures are aligned to a fullword and occupy `(sizeof(struct X)+3)/4` fullwords, with any padding at the end. A structure that is smaller than a word is left-justified within its word or register. Larger structures can occupy multiple registers and may be passed partly in storage and partly in registers.
- Other aggregate values, including Pascal records, are passed “val-by-ref”. That is, the compiler actually passes their address and arranges for a copy to be made in the invoked program.
- A procedure or function pointer is passed as a pointer to the routine’s function descriptor; its first word contains its entry point address. (See “Pointers to Functions” on page 365 for more information.)

In a 64-bit Environment

- All non-floating-point values require one doubleword that is doubleword aligned.
- Each single-precision (**REAL(4)**) value and each double-precision (**REAL(8)**) value occupies one doubleword in the list. Each extended-precision (**REAL(16)**) value occupies two successive doublewords in the list.
- A **COMPLEX** value occupies twice as many doublewords as a **REAL** value with the same kind type parameter.
- In Fortran and C, structure values appear in successive words as they would anywhere in storage, satisfying all appropriate alignment requirements. Structures are aligned to a doubleword and occupy `(sizeof(struct X)+7)/8` doublewords, with any padding at the end. A structure that is smaller than a doubleword is left-justified within its doubleword or register. Larger structures can occupy multiple registers and may be passed partly in storage and partly in registers.
- Other aggregate values, including Pascal records, are passed “val-by-ref”. That is, the compiler actually passes their address and arranges for a copy to be made in the invoked program.
- A procedure or function pointer is passed as a pointer to the routine’s function descriptor; its first word contains its entry point address. (See “Pointers to Functions” on page 365 for more information.)

Argument Passing Rules (by Value)

From the following illustration, we state these rules:

- In a 32-bit environment, the parameter list is a conceptually contiguous piece of storage that contains a list of words. For efficiency, the first 8 words of the list are not actually stored in the space that is reserved for them but are passed in GPR3-GPR10. Further, the first 13 floating-point value parameters are passed in FPR1-FPR13. Those beyond the first 8 words of the parameter list are also in storage. Those within the first 8 words of the parameter list have GPRs reserved for them, but they are not used.
- In a 64-bit environment, the preceding information holds true if references to words are replaced with doublewords.
- If the called procedure treats the parameter list as a contiguous piece of storage (for example, if the address of a parameter is taken in C), the parameter registers are stored in the space reserved for them in the stack.
- A register image is stored on the stack.
- The argument area ($P_1 ... P_n$) must be large enough to hold the largest parameter list.

Here is an example of a call to a function:

```
f(%val(l1), %val(l2), %val(l3), %val(d1), %val(f1),
          %val(c1), %val(d2), %val(s1), %val(cx2))
```

where:

l denotes integer(4) (fullword integer)
d denotes real(8) (double precision)
f denotes real(4) (real)
s denotes integer(2) (halfword integer)
c denotes character (one character)
cx denotes complex(8) (double complex)

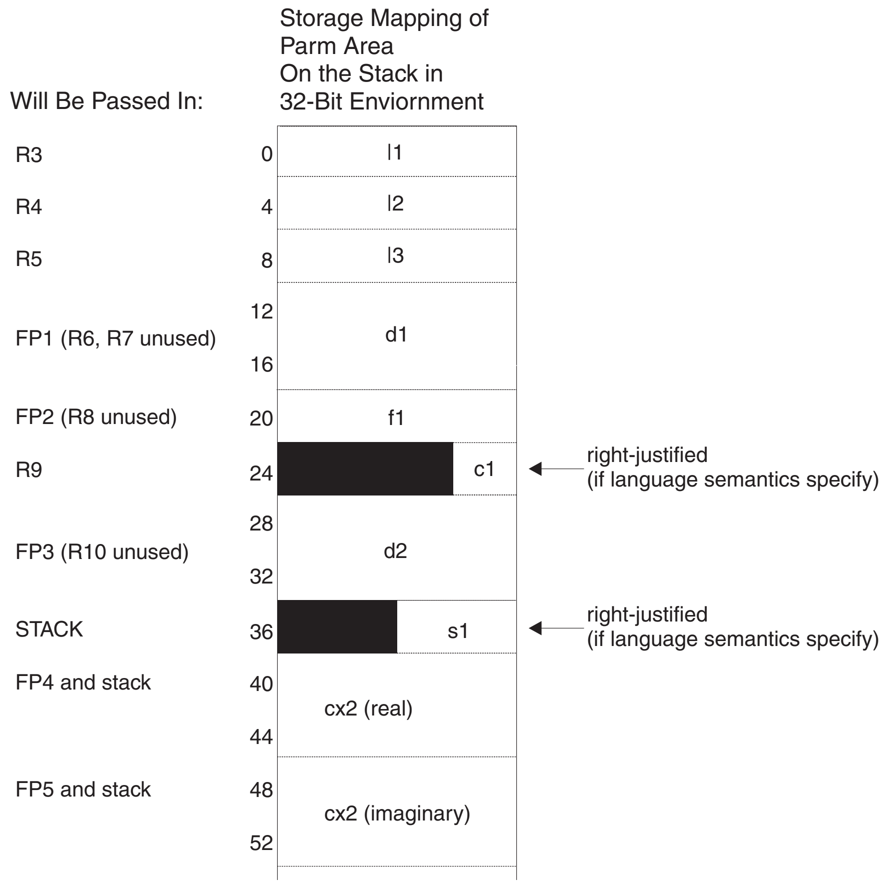


Figure 4. Storage Mapping of Parm Area On the Stack in 32-Bit Environment

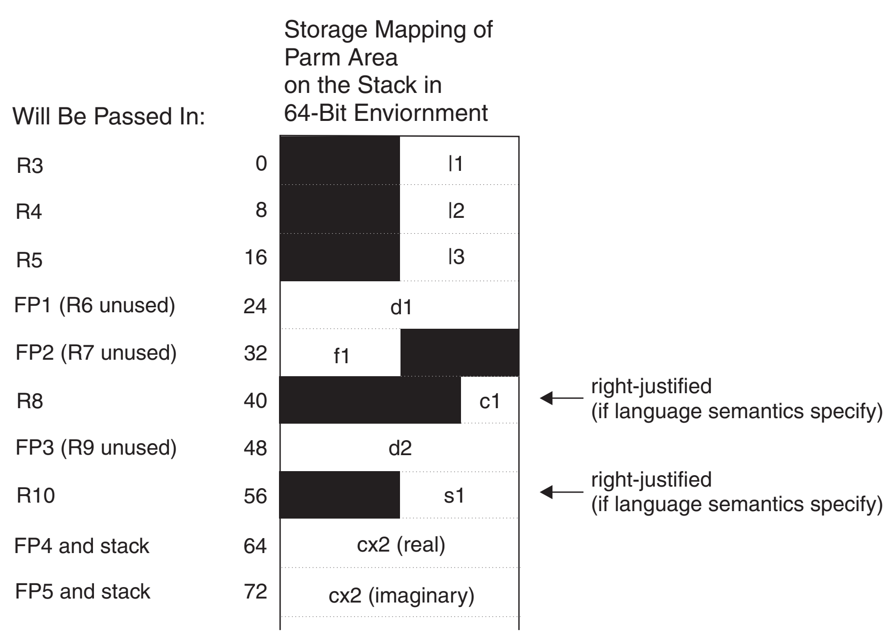


Figure 5. Storage Mapping of Parm Area On the Stack in 64-Bit Environment

Order of Arguments in Argument List

The argument list is constructed in the following order. Items in the same bullet appear in the same order as in the procedure declaration, whether or not argument keywords are used in the call.

- All addresses or values (or both) of actual arguments [4]
- "Present" indicators for optional arguments that are passed by value
- Length arguments for strings [4]

Linkage Convention for Function Calls

A routine has two symbols associated with it: a function descriptor (*name*) and an entry point (*.name*). When a call is made to a routine, the program branches to the entry point directly. Excluding the loading of parameters (if any) in the proper registers, compilers expand calls to functions to the following two-instruction sequence:

```
BL    .foo               # Branch to foo
ORI R0,R0,0x0000         # Special NOP
```

The linker does one of two things when it encounters a **BL** instruction:

1. If `foo` is imported (not in the same object module), the linker changes the **BL** to `.foo` to a **BL** to `.glink` (global linkage routine) of `foo` and inserts the `.glink` into the object module. Also, if a **NOP** instruction (`ORI R0,R0,0x0000`) immediately follows the **BL** instruction, the linker replaces the **NOP** instruction with the **LOAD** instruction `L R2, 20(R1)`.
2. If `foo` is bound in the same object module as its caller and a **LOAD** instruction `L R2,20(R1)` for 32-bit and `L R2,40(R1)` for 64-bit, or `ORI R0,R0,0` immediately follows the **BL** instruction, the linker replaces the **LOAD** instruction with a **NOP** (`ORI R0,R0,0`).

Note: For any export, the linker inserts the procedure's descriptor into the object module.

Pointers to Functions

A function pointer is a data type whose values range over procedure names. Variables of this type appear in several programming languages, such as C and Fortran. In Fortran, a dummy argument that appears in an **EXTERNAL** statement is a function pointer. Fortran provides support for the use of function pointers in contexts such as the target of a call statement or an actual argument of such a statement.

A function pointer is a fullword quantity that is the address of a function descriptor. The function descriptor is a 3-word object. The first word contains the address of the entry point of the procedure. The second has the address of the TOC of the object module in which the procedure is bound. The third is the environment pointer for some non-Fortran languages. There is only one function descriptor per entry point. It is bound into the same object module as the function it identifies if the function is external. The descriptor has an external name, which is the same as the function name but with a different storage class that uniquely identifies it. This descriptor name is used in all import or export operations.

4. There may be other items in this list during Fortran-Fortran calls. However, they will not be visible to non-Fortran procedures that follow the calling rules in this section.

Function Values

Functions return their values according to type:

- **INTEGER** and **LOGICAL** of kind 1, 2, and 4 are returned (right justified) in R3.
- In 32-bit mode, **INTEGER** and **LOGICAL** of kind 8 are returned in R3 and R4.
- In 64-bit mode, **INTEGER** and **LOGICAL** of kind 8 are returned in R3.
- **REAL** of kind 4 or 8 are returned in FP1. **REAL** of kind 16 are returned in FP1 and FP2.
- **COMPLEX** of kind 4 or 8 are returned in FP1 and FP2. **COMPLEX** of kind 16 are returned in FP1-FP4.
- Character strings are returned in a buffer allocated by the caller. The address and the length of this buffer are passed in R3 and R4 as hidden parameters. The first explicit parameter word is in R5, and all subsequent parameters are moved to the next word.
- Structures are returned in a buffer that is allocated by the caller. The address is passed in R3; there is no length. The first explicit parameter is in R4.

The Stack Floor

The stack floor is a system-defined address below which the stack cannot grow. All programs in the system must avoid accessing locations in the stack segment that are below the stack floor.

All programs must maintain other system invariants that are related to the stack:

- No data is saved or accessed from an address lower than the stack floor.
- The stack pointer is always valid. When the stack frame size is more than 32 767 bytes, you must take care to ensure that its value is changed in a single instruction. This step ensures that there is no timing window where a signal handler would either overlay the stack data or erroneously appear to overflow the stack segment.

Stack Overflow

The linkage convention requires no explicit inline check for overflow. The operating system uses a storage protection mechanism to detect stores past the end of the stack segment.

Prolog and Epilog

On entry to a procedure, you might have to do some or all of the following steps:

1. Save the link register at offset 8 for 32-bit environments (or offset 16 for 64-bit environments) from the stack pointer if necessary.
2. If you use any of the CR bits 8-23 (CR2, CR3, CR4, CR5), save the CR at displacement 4 for 32-bit environments (or displacement 8 for 64-bit environments) from the current stack pointer.
3. Save any nonvolatile FPRs that are used by this procedure in the caller’s FPR save area. You can use a set of routines: **_savef14**, **_savef15**, ... **_savef31**.
4. Save all nonvolatile GPRs that are used by this procedure in the caller’s GPR save area.
5. Store back chain and decrement stack pointer by the size of the stack frame. Note that if a stack overflow occurs, it will be known immediately when the store of the back chain is done.

On exit from a procedure, you might have to perform some or all of the following steps:

1. Restore all GPRs saved.
2. Restore stack pointer to the value it had on entry.
3. Restore link register if necessary.
4. Restore bits 8-23 of the CR if necessary.
5. If you saved any FPRs, restore them using **_restf***n*, where *n* is the first FPR to be restored.
6. Return to caller.

Traceback

The compiler supports the traceback mechanism, which symbolic debuggers need to unravel the call or return stack. Each object module has a traceback table in the text segment at the end of its code. This table contains information about the object module, including the type of object module, as well as stack frame and register information.

Related Information: You can make the traceback table smaller or remove it entirely with the "-qtbtable Option" on page 249.

THREADLOCAL Common Blocks and ILC with C

Fortran **THREADLOCAL** common blocks are implemented using the thread-specific data facilities that are defined by the POSIX pthreads library. For additional information about thread-specific data areas, please refer to AIX documentation on threads programming.

Internally, the storage for the thread-specific common block is allocated dynamically by the Fortran run-time library. The Fortran run-time library maintains a control structure that holds information about the common block. This control area is an external structure whose name is the name of the common block.

For example, if you declare a common block in Fortran as the following:

```
        common /myblock/ i
!ibm*   threadlocal /myblock/
```

the Fortran compiler creates an external structure (or common area) that is named *myblock*, which contains control information about the thread-specific common block.

The control structure has the following layout and would be coded as such in C:

```
typedef struct {
   pthread_key_t key;
   int flags;
   void *unused_1;
   int unused_2;
} FORT_LOCAL_COMMON;
extern FORT_LOCAL_COMMON myblock;
```

The ″key″ field is a unique identifier that describes a threadlocal data area. Every threadlocal common block has its own key. The ″flags″ field indicates whether a key has been obtained for the common block. Within a C function, you should use the ″key″ in the control block in a call to **pthread_getspecific** to obtain the thread-specific address of the threadlocal common area.

Example

```
! Example 1:  "fort_sub" is invoked by multiple threads.  This is an invalid example
! because "fort_sub" and "another_sub" both declare /block/ to be THREADLOCAL.
! They intend to share the common block, but they are executed by different threads.

SUBROUTINE fort_sub()
  COMMON /block/ j
  INTEGER :: j
  !IBM* THREADLOCAL /block/        ! Each thread executing fort_sub
                                   ! obtains its own copy of /block/.
  INTEGER a(10)

  ...
  !IBM* INDEPENDENT
  DO index = 1,10
    CALL another_sub(a(i))
  END DO
  ...

END SUBROUTINE fort_sub

SUBROUTINE another_sub(aa)        ! Multiple threads are used to execute another_sub.
  INTEGER aa
  COMMON /block/ j                ! Each thread obtains a new copy of the
  INTEGER :: j                    ! common block:  /block/.
  !IBM* THREADLOCAL /block/
  ...
  aa = j                          ! The value of 'j' is undefined.
END SUBROUTINE another_sub
```

For more information, see the **THREADLOCAL** directive, in the *XL Fortran Enterprise Edition for AIX Language Reference*.

Problem Determination and Debugging

This section describes some methods you can use for locating and fixing problems in compiling or executing your programs.

Related Information: You might encounter a number of potential problems when moving from previous versions of XL Fortran to XL Fortran Version 9. "Avoiding or Fixing Upgrade Problems" on page 25 summarizes these potential problems.

Understanding XL Fortran Error Messages

Most information about potential or actual problems comes through messages from the compiler or application program. These messages are written to the standard error output stream.

Error Severity

Compilation errors can have the following severity levels, which are displayed as part of some error messages:

U An unrecoverable error. Compilation failed because of an internal compiler error.

S A severe error. Compilation failed due to one of the following:
- Conditions exist that the compiler could not correct. An object file is produced; however, you should not attempt to run the program.
- An internal compiler table has overflowed. Processing of the program stops, and XL Fortran does not produce an object file.
- An include file does not exist. Processing of the program stops, and XL Fortran does not produce an object file.
- An unrecoverable program error has been detected. Processing of the source file stops, and XL Fortran does not produce an object file. You can usually correct this error by fixing any program errors that were reported during compilation.

E An error that the compiler can correct. The program should run correctly.

W Warning message. It does not signify an error but may indicate some unexpected condition.

L Warning message that was generated by one of the compiler options that check for conformance to various language levels. It may indicate a language feature that you should avoid if you are concerned about portability.

I Informational message. It does not indicate any error, just something that you should be aware of to avoid unexpected behavior.

Notes:
1. The message levels **S** and **U** indicate a compilation failure.
2. The message levels **I**, **L**, **W**, and **E** indicate that compilation was successful.

By default, the compiler stops without producing output files if it encounters a severe error (severity S). You can make the compiler stop for less severe errors by specifying a different severity with the **-qhalt** option. For example, with **-qhalt=e**, the compiler stops if it encounters any errors of severity E or higher severity. This technique can reduce the amount of compilation time that is needed to check the syntactic and semantic validity of a program. You can limit low-severity messages without stopping the compiler by using the **-qflag** option. If you simply want to prevent specific messages from going to the output stream, see "-qsuppress Option" on page 245.

Compiler Return Code

The compiler return codes and their respective meanings are as follows:

0 The compiler did not encounter any errors severe enough to make it stop processing a compilation unit.

1 The compiler encountered an error of severity E or *halt_severity* (whichever is lower). 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.

Run-Time Return Code

If an XLF-compiled program ends abnormally, the return code to the operating system is 1.

Note: This is a change from XL Fortran Version 2, which used a value of 232 in this case.

If the program ends normally, the return code is 0 (by default) or is **MOD**(*digit_string*,256) if the program ends because of a **STOP** *digit_string* statement.

Understanding XL Fortran Messages

In addition to the diagnostic message issued, the source line and a pointer to the position in the source line at which the error was detected are printed or displayed if you specify the **-qsource** compiler option. If **-qnosource** is in effect, the file name, the line number, and the column position of the error are displayed with the message.

The format of an XL Fortran diagnostic message is:

```
>>—15—cc—-—nnn— ——————————————————————————message_text——————————————><
                └(—severity_letter—) ┘
```

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
24	Indicates a VAST preprocessor message
25	Indicates a run-time message from an XL Fortran application program
26	Indicates a KAP preprocessor message
85	Indicates a loop-transformation message
86	Indicates an interprocedural analysis (IPA) message

nnn Is the message number

severity_letter Indicates how serious the problem is, as described in the preceding section

'*message text*' Is the text describing the error

Limiting the Number of Compile-Time Messages

If the compiler issues many low-severity (**I** or **W**) messages concerning problems you are aware of or do not care about, use the **-qflag** option or its short form **-w** to limit messages to high-severity ones:

```
# E, S, and U messages go in listing; U messages are displayed on screen.
xlf95 -qflag=e:u program.f

# E, S, and U messages go in listing and are displayed on screen.

xlf95 -w program.f
```

Selecting the Language for Messages

By default, XL Fortran comes with messages in U.S. English only. You can also order translated message catalogs:
- Compiler messages in Japanese
- Run-time messages in Japanese

If compile-time messages are appearing in U.S. English when they should be in another language, verify that the correct message catalogs are installed and that the **LANG**, **LC_MESSAGES**, and/or **LC_ALL** environment variables are set accordingly.

If a run-time message appears in the wrong language, also ensure that your program calls the **setlocale** routine.

Related Information: See “Environment Variables for National Language Support” on page 13 and “Selecting the Language for Run-Time Messages” on page 50.

To determine which XL Fortran message catalogs are installed, use the following commands to list them:

```
lslpp -f 'xlfcmp.msg.*' # compile-time messages
lslpp -f 'xlfrte.msg.*' # run-time messages
```

The file names of the message catalogs are the same for all supported international languages, but they are placed in different directories.

Note: When you run an XL Fortran program on a system without the XL Fortran message catalogs, run-time error messages (mostly for I/O problems) are not displayed correctly; the program prints the message number but not the associated text. To prevent this problem, copy the XL Fortran message catalogs from **/usr/lpp/xlf/bin/default_msg** to a directory that is part of the **NLSPATH** environment-variable setting on the execution system.

Fixing Installation or System Environment Problems

If individual users or all users on a particular machine have difficulty running the compiler, there may be a problem in the system environment. Here are some common problems and solutions:

xlf90: not found
xlf90_r: not found
xlf90_r7: not found
xlf95: not found
xlf95_r: not found
xlf95_r7: not found
xlf: not found
xlf_r: not found
xlf_r7: not found
f77: not found
fort77: not found
f90: not found
f95: not found
Symptom: The shell cannot locate the command to execute the compiler.
Solution: Make sure that your **PATH** environment variable includes the directory **/usr/bin**. If the compiler is properly installed, the commands you need to execute it are in this directory.

Could not load program *program*
Error was: not enough space
Killed
Symptom: The system cannot execute the compiler or an application program at all.
Solution: Set the storage limits for stack and data to "unlimited" for users who experience this problem. For example, as superuser you can set both your hard and soft limits with these **ksh** commands:

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

Because non-superusers are not completely free to give themselves unlimited limits, if you are a superuser you may find it more convenient to edit the file **/etc/security/limits** 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 program *program*
Could not load library *library_name*.**a**
[*object_name***]**
Error was: no such file or directory
Solution: Make sure the XL Fortran libraries are installed in **/usr/lib**, or set the **LIBPATH** environment variable to include the directory where **libxlf90.a** is installed if it is in a different directory. See "LIBPATH:Setting Library Search Paths" on page 14 for details of this environment variable.

Symptom: Messages from the compiler or an XL Fortran application program are displayed in the wrong language.
Solution: Set the appropriate national language environment. You can set the national language for each user with the command **smit chlang**. 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 13 provides details.

Symptom: A compilation fails with an I/O error.
Solution: Increase the size of the **/tmp** filesystem, or set the environment variable **TMPDIR** to the path of a filesystem that has more free space.
Explanation: The object file may have grown too large for the filesystem that holds it. The cause could be a very large compilation unit or initialization of all or part of a large array in a declaration.

Symptom: There are too many individual makefiles and compilation scripts to easily maintain or track.
Solution: Add stanzas to the configuration file, and create links to the compiler by using the names of these stanzas. By running the compiler with different command names, you can provide consistent groups of compiler options and other configuration settings to many users.

Fixing Compile-Time Problems

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

Duplicating Extensions from Other Systems

Some ported programs may cause compilation problems because they rely on extensions that exist on other systems. XL Fortran supports many extensions like these, but some require compiler options to turn them on. See "Options for Compatibility" on page 79 for a list of these options and "Porting Programs to XL Fortran" on page 397 for a general discussion of porting.

Isolating Problems with Individual Compilation Units

If you find that a particular compilation unit requires specific option settings to compile properly, you may find it more convenient to apply the settings in the source file through an **@PROCESS** directive. Depending on the arrangement of your files, this approach may be simpler than recompiling different files with different command-line options.

Compiling with Thread-safe Commands

Thread-safe invocation commands, like **xlf_r** or **xlf90_r**, for example, use different search paths and call different modules than the non thread-safe invocations. Your programs should account for the different usages. Programs that compile and run successfully for one environment may produce unexpected results when compiled and run for a different use. The configuration file, **xlf.cfg**, shows the paths, libraries, and so on for each invocation command. (See "Customizing the Configuration File" on page 15 for a sample configuration file and an explanation of its contents.)

Running out of Machine Resources

If the operating system runs low on resources (page space or disk space) while one of the compiler components is running, you should receive one of the following messages:

```
1501-229 Compilation ended because of lack of space.
1501-224 fatal error in /usr/lpp/xlf/bin/xlfentry: signal 9 received.
1517-011 Compilation ended.  No more system resources available.
Killed.
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 *AIX General Concepts and Procedures* 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** or **bind** commands. A frequently encountered error and its solution are listed here for your convenience:

0706-317 **ERROR: Undefined or unresolved symbols detected:**

Symptom: A program cannot be linked because of unresolved references.

Explanation: Either needed object files or libraries are not being used during linking, there is an error in the specification of one or more external names, or there is an error in the specification of one or more procedure interfaces.

Solution: You may need to do one or more of the following actions:

- Compile again with the **-bloadmap** 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.
- Use the linker **-brename** option on the compiler command line to change the names of some symbols at link time.

Fixing Run-Time Problems

XL Fortran issues error messages during the running of a program in either of the following cases:

- XL Fortran detects an input/output error. "Setting Run-Time Options" on page 51 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 **dbx**.

  The causes for run-time exceptions are listed in "XL Fortran Run-Time Exceptions" on page 66.

You can investigate errors that occur during the execution of a program by using a symbolic debugger, such as **dbx**.

Duplicating Extensions from Other Systems

Some ported programs may not run correctly if they rely on extensions that are found on other systems. XL Fortran supports many such extensions, but you need to turn on compiler options to use some of them. See "Options for Compatibility" on page 79 for a list of these options and "Porting Programs to XL Fortran" on page 397 for a general discussion of porting.

Mismatched Sizes or Types for Arguments

To detect arguments of different sizes or types, which might produce incorrect execution and results, you can compile with the **-qextchk** option. This option warns you of any problems at link time.

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 Enterprise Edition for AIX 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 Enterprise Edition for AIX Language Reference*.

If you have installed the XL Fortran run-time message catalog on the system on which the program is executing, a message number and message text are issued to the terminal (standard error) for certain I/O errors. If you have specified **IOMSG** on the input/output statement, the **IOMSG** variable is assigned the error message text if an error is detected, or the content of **IOMSG** variable is not changed. If this catalog is not installed on the system, only the message number appears. Some of the settings in "Setting Run-Time Options" on page 51 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 **ksh**, or through the **smit** command.

Tracebacks and Core Dumps

If a run-time exception occurs and an appropriate exception handler is installed, a message and a traceback listing are displayed. Depending on the handler, a core file might be produced as well. You can then use a debugger to examine the location of the exception.

To produce a traceback listing without ending the program, call the **xl__trbk** procedure:

```
IF (X .GT. Y) THEN    ! X > Y indicates that something is wrong.
  PRINT *, 'Error - X should not be greater than Y'
  CALL XL__TRBK       ! Generate a traceback listing.
  X = 0               ! The program continues.
END IF
```

See "Installing an Exception Handler" on page 298 for instructions about exception handlers and "XL Fortran Run-Time Exceptions" on page 66 for information about the causes of run-time exceptions.

Debugging a Fortran 90 or Fortran 95 Program

XL Fortran includes a technology preview of the IBM Distributed Debugger, a client-server debugger, which you can use to help debug your programs. The Distributed Debugger can debug programs running on systems accessible through a network connection as well as debug programs running on your workstation.

For instructions on using your chosen debugger, consult the online help within the debugger or its documentation.

Always specify the **-g** option when compiling programs for debugging.

Related information: See "Options for Error Checking and Debugging" on page 75.

A Sample dbx Session for an XL Fortran Program

You can debug XL Fortran programs with any **dbx**-compatible symbolic debugger. For background information on **dbx**, see the *AIX General Concepts and Procedures* document . For information on **dbx** subcommands, see the *AIX Commands Reference*.

The following example represents a typical XL Fortran problem that you may be able to resolve through **dbx**. Although it demonstrates only a small subset of **dbx** features and uses memory-allocation techniques made obsolete by Fortran 90/Fortran 95 allocatable arrays, it can serve as an introduction if you have not used this debugger before.

Problem with Dynamic Memory Allocation

The following program tries to allocate an array at run time by using the AIX system subroutine **malloc**. When you use the following command to compile the program and then run the program, the program produces a core dump:

```
xlf95 -qddim testprog.f -o testprog
```

At this point, you may be wondering whether the C **malloc** routine is working correctly or whether this is the right way to allocate an array in a main program when the dimensions are not known until run time.

```
       program main

pointer(p, array(nvar,nrec))
real*8 array

nvar = 2
nrec = 3
p = malloc(nvar*nrec*8)

call test_sub(array, nvar, nrec)

end

subroutine test_sub(array, nvar, nrec)

dimension array(nvar, nrec)

array(1,1) = 1.
array(2,1) = 2.
array(1,2) = 3.
array(2,2) = 4.
array(1,3) = 5.
array(2,3) = 6.

write(*, 100) array(1,1), array(2,1), array(1,2),
  1         array(2,2), array(1,3), array(2,3)
100     format(//t2,f4.1/t2,f4.1/t2,f4.1/t2,f4.1/
  1         t2,f4.1/t2,f4.1)

return
end
```

You might go through the debugging process as follows:

1. Compile the program with the **-g** option, to allow debugging under **dbx**:

```
 -> xlf95  -qddim -g testprog.f -o testprog
** main   === End of Compilation 1 ===
** test_sub   === End of Compilation 2 ===
1501-510  Compilation successful for file testprog.f.
```

2. Run the program to verify the problem and create a core dump:

```
 -> testprog
Segmentation fault(coredump)
 ->
```

3. Find out where in the program the core dump occurs:

```
 -> dbx testprog core
dbx version 3.1 for AIX.
Type 'help' for help.
reading symbolic information ...
[using memory image in core]

segmentation violation in test_sub at line 21 in file "testprog.f"
   21             array(1,1) = 1.
(dbx)
```

4. Use the **where** command to get a traceback of the calls that led to that point in the program:

```
(dbx) where
test_sub(array = (...), nvar = warning: Unable to access address 0x200aee94
 from core
-1, nrec = warning: Unable to access address 0x200aee98 from core
-1), line 21 in "testprog.f"
main(), line 12 in "testprog.f"
(dbx)
```

main calls **test_sub** at line 12. The warning indicates that a problem occurs while evaluating the arguments for this call.

5. Look at the value of the first argument of the array:

```
(dbx) print array(1,1)
reference through nil pointer
(dbx)
```

This suggests that **array** does not have a value assigned. To verify that possibility, try to look at the address of an element in the array:

```
(dbx) p &array(1,1)
(nil)
(dbx)
```

It seems that XL Fortran has not allocated the space for the array. To verify that it has not, print the value of the pointer that points to the array:

```
(dbx) print p
warning: Unable to access address 0x200aee90 from core
0xffffffff
```

6. To find out what happens to **p** during execution, restart the program and trace the use of **p**:

```
(dbx) stop in main
[1] stop in main
(dbx) run
[1] stopped in main at line 7 in file "testprog.f"
    7            nvar = 2
(dbx) trace p
[3] trace p
(dbx) cont
initially (at line 8 in "testprog.f"):  p = nil

segmentation violation in test_sub at line 21 in file "testprog.f"
   21            array(1,1) = 1.
(dbx) p p
nil
(dbx)
```

Because **p** is never set to a valid value, something must be wrong with the line that allocates space for the array:

```
     9           p = malloc(nvar*nrec*8)
```

7. The next step is to research why the call to **malloc** does not work. Because **malloc** is a C function, you should read "Interlanguage Calls" on page 345 for background knowledge and specific guidelines.

   When you read that section, you find that calls to C functions require arguments to be passed by value, rather than by reference. To fix the problem in this sample program, replace the line:

   ```
   p = malloc(nvar*nrec*8)
   ```

   with the line:

   ```
   p = malloc(%val(nvar*nrec*8))
   ```

8. Compiling and running the fixed program (**solution.f**) again produces the correct result:

```
 -> xlf95  -qddim -g solution.f -o solution
** main   === End of Compilation 1 ===
** test_sub   === End of Compilation 2 ===
1501-510  Compilation successful for file solution.f.
 -> solution


  1.0
  2.0
  3.0
  4.0
  5.0
  6.0
```

9. It might be informative to trace the execution of the corrected program:

```
 -> dbx solution
dbx version 3.1 for AIX.
Type 'help' for help.
Core file program (testprog) does not match current program (core ignored)
reading symbolic information ...
(dbx) trace p
[1] trace p
(dbx) run
initially (at line 7 in "solution.f"):  p = nil
after line 9 in "solution.f":   p = 0x200af100


  1.0
  2.0
  3.0
  4.0
  5.0
  6.0

execution completed
(dbx)
```

To check whether the values of **p** and **array** are appropriate, turn off the trace:

```
(dbx) status
[1] trace p
(dbx) delete all
(dbx) status
(dbx)
```

Then set new break points and run through the program again. Notice that the address of **array(1,1)** is the same as the contents of **p(0x200af100)**, as expected:

```
(dbx) stop at 9
[11] stop at "solution.f":9
(dbx) run
[11] stopped in main at line 9 in file "solution.f"
    9            p = malloc(%val(nvar*nrec*8))
(dbx) p p
nil
(dbx) next
stopped in main at line 12 in file "solution.f"
   12            call test_sub(array, nvar, nrec)
(dbx) p p
0x200af100     <-------------
(dbx)
(dbx) step      /* Notice we use step to step into subroutine test_sub. */
stopped in test_sub at line 21 in file "solution.f"
   21            array(1,1) = 1.
(dbx) p &array(1,1)
0x200af100     <---------------
(dbx) next
stopped in test_sub at line 22 in file "solution.f"
   22            array(2,1) = 2.
(dbx) p array(1,1)
1.0
(dbx)
```

Using Debug Memory Routines for XL Fortran

The XL Fortran compiler contains two libraries that are geared to various memory-allocation facilities. These libraries include:

libhmd.a A library that provides debug versions of memory-management routines.

libhm.a A non-debug library that provides replacement routines for **malloc**, **free**, and so on. These routines are faster than the usual AIX versions. In addition, this library contains a few new library routines to provide additional facilities for memory management and production-level heap error checking.

The library of most interest to Fortran users is **libhmd.a**. See “The libhmd.a Library” on page 383 for additional details. If you are installing an application built with these libraries in an environment that does not have XL Fortran installed, you may need to include the library **libhu.a** as well. This is because some routines in **libhmd.a** and **libhm.a** are dependent on routines in **libhu.a**.

The libhm.a Library

libhm.a provides fast replacement routines for the following **libc.a** procedures: **malloc**, **calloc**, **realloc**, **free**, **strdup**, **mallopt**, and **mallinfo**. The interfaces to these routines are exactly the same as the interfaces to the standard system routines, so a user need only link in **libhm.a** before the system libraries to make use of them.

In addition, the following library routines that are provided in **libhm.a** are available to Fortran users: **_heapchk** and **_heapset**. These routines provide production-level services that assist in maintaining consistent and correct heap storage.

Note that you cannot use the **-qextname** compiler option with programs that use these facilities. In other words, since these library routines are "system-like" routines that are not Fortran-specific, we do not provide "_" versions of the routines in our library.

The following table describes the additional routines that you can use from **libhm.a**:

<table>
<tr><th>C Function prototype</th><th>Fortran usage example</th><th>Description</th></tr>
<tr><td><code>int _heapchk(void);</code></td><td><pre>integer(4) _heapchk,
  retc
retc = _heapchk()</pre></td><td>Does consistency checking for all allocated and freed objects on the heap.

Return values:

0 The heap is consistent.

1 Reserved.

2 Heap errors have occurred.</td></tr>
<tr><td><pre>int _heapset
  (unsigned int fill);</pre></td><td><pre>integer(4) _heapset,
  retc
integer(4) fill /1/

retc =
  _heapset(%val(fill))</pre></td><td>_heapset checks the heap for consistency (similar to _heapchk). It then sets each byte of any non-reserved freed storage to the value of fill. The value of fill must be an integer in the range of 0-255.

Using _heapset can help a user locate problems where a program continues to use a freed pointer to an object.

Return values:

0 The heap is consistent.

1 Reserved.

2 Heap errors have occurred.</td></tr>
</table>

Examples:

Example 1: Using _heapchk to test for heap errors

```
program tstheapchk
pointer (p,pbased),(q,qbased)
integer pbased,qbased

integer(4) _heapchk,retcode

p = malloc(%val(4))
pbased = 10

! Decrement the pointer and store into
       ! memory we do not own.
q = p-4;
qbased = 10

retcode = _heapchk()
if (retcode .ne. 0) call abort()
```

```
! Expected return code is: 2.  Program will be aborted.

call free(%val(p))
end
```

Example 2: Using _heapset

```
       program tstheapset
       pointer (p,based)
       integer*1 based(1000)
       integer _heapset,retcode

       p = malloc(%val(1000))
       based = 1
       print *,based(450:500)

       call free(%val(p))

       retcode = _heapset(%val(2))
       print *,based(450:500)
       end
```

```
Output:
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
```

Example 3: Using _heapchk to test for heap errors with ALLOCATE and DEALLOCATE

```
      program tstheapchk
        integer, allocatable :: a(:)
        integer(4) :: retcode
        integer(4), external :: _heapchk

        allocate(a(1:5))

       ! Store outside the bounds of allocated memory.
        a(-5:10) = 17

        retcode = _heapchk()
        if (retcode /= 0) call abort()
        print *, retcode

        deallocate(a)
     end program tstheapchk
```

Example 4: Using _heapset with memory managed with ALLOCATE and DEALLOCATE

```
     program tstheapset
        integer(1), pointer :: p1(:), p2(:)
        integer(4) :: retcode
        integer(4), external :: _heapset

        allocate(p1(1:10))
        p2 => p1(6:10)
        p1 = 1
        print *, p2

        deallocate(p1)

        retcode = _heapset(%val(2))
        print *, p2
     end program tstheapset
```

```
Output:
 1 1 1 1 1
 2 2 2 2 2
```

The libhmd.a Library

The facilities that are provided with **libhmd.a** include:

- Memory leak reporting that indicates where the allocation of non-freed storage occurred and that displays partial contents of the area that is not freed.
- Memory error detection that includes:
  - Freeing the same location multiple times
  - Overwriting the end of an allocated object (note that the **-qcheck** compiler option available with XL Fortran already provides much of this functionality)
  - Reading data from or writing data to a freed object
  - Freeing an invalid pointer

You obtain access to this functionality when you link in the **libhmd.a** library prior to the system libraries. References to **malloc**, **realloc**, and **free** can be explicit, or you can obtain heap debugging for memory allocated and deallocated via the **ALLOCATE** and **DEALLOCATE** statements. To obtain source line number information in the output that the debug library produces, you should compile with the **-g** compiler option.

Note that you cannot specify the **-qextname** compiler option with programs that use these facilities.

The following shows the external interface and description of the procedures that are provided. Note that the external interfaces and functionality of **malloc**, **free**, **calloc**, **realloc**, and **strdup** are not shown in this table, since they have not changed.

C Function prototype	Fortran usage example	Description

`void _dump_allocated`
`  (int size);`
	`integer(4) :: size=4`
	`call _dump_allocated &`
	`   (%val(size))`
		This routine prints information to stderr about each memory block that is currently allocated or was allocated using the debug memory management routines. *size* indicates how many bytes of each memory block are to be printed, as follows:
		Negative size All bytes are displayed.
		0 size No bytes are displayed.
		Positive size Specified number of bytes are displayed.

C Function prototype	Fortran usage example	Description

`void _dump_allocated_delta`
`    (int size);`
	`integer(4) :: size=4`
	`call _dump_allocated_delta &`
	`   (%val(size))`
		This routine prints information to stderr about each memory block that is currently allocated or was allocated using the debug memory management routines since the last call to *_dump_allocated* or *_dump_allocated_delta*. *size* indicates how many bytes of each memory block are to be printed, as follows:
		Negative size All bytes are displayed.
		0 size No bytes are displayed.
		Positive size Specified number of bytes are displayed.

`void heap_check(void);`	`call _heap_check()`	This routine does consistency checking for memory blocks that have been allocated using the debug memory-management routines. It checks that your program has not overwritten freed storage or memory outside the bounds of allocated blocks. All of the debug memory-allocation routines (debug version of malloc, and so on) invoke *heap_check* automatically. It may also be invoked explicitly in areas of code where a user believes there may be memory problems. Calling *heap_check* frequently can increase the memory requirements and affect the performance of a program. The **HD_SKIP** environment variable may be used to control how often the debug memory procedures check the heap. Note that the errors are detected when the *heap_check* routine is invoked, not at the point of error in the program.

Environment Variables

The debug libraries support the following environment variables:

HD_SKIP=*increment* **[,***start***]** Control how often *heap_check* is invoked from the debug versions of the memory-management routines. *increment* indicates how often you want the debug functions to check the heap. *start* specifies that the skipping of heap checks should begin after the debug memory routines have been called a certain number of times. The default values for *increment* and *start* are 1 and 0, respectively.

HD_FILL
When this environment variable is exported, the debug versions of *malloc* and *realloc* set the memory allocated to a byte pattern of 0xAA.

HD_STACK=*n*
n specifies how many procedures should appear in the call chain that the debug memory routines produce. The default is 10, or it is the number of routines in the call chain, if fewer than 10.

For example:

```
       export HD_SKIP=10
       ! Every 10th debug memory function calls heap_check.

 export HD_SKIP=100,10
 ! After 100 calls to debug memory functions, every 10th call
       ! will result in a call to heap_check.
```

Examples:

Example 1: Memory leak detection

```
   pointer (p,a),(p2,b),(p3,c)               !  1
   character a(4)                            !  2
   integer b,c                               !  3
          !  4
   p = malloc(%val(4))                       !  5
   a(1) = 'a'                                !  6
   a(2) = 'b'                                !  7
   a(3) = 'c'                                !  8
   a(4) = 'd'                                !  9
          !  10
   p2 = malloc(%val(4))                      !  11
   b = 1                                     !  12
          !  13
   call _dump_allocated(%val(4))             !  14
          !  15
   p3 = malloc(%val(4))                      !  16
   c = 2                                     !  17
          !  18
   call _dump_allocated_delta(%val(4))       !  19
   end                                       !  20
```

Output:

```
1546-515 -----------------------------------------------------------------
1546-516                  START OF DUMP OF ALLOCATED MEMORY BLOCKS
1546-515 -----------------------------------------------------------------
1546-518 Address: 0x20000DE0      Size: 0x00000004 (4)
    _int_debug_umalloc + 32C
        _debug_umalloc + 44
   _dbg_umalloc + 18
  _umalloc_init + 30
         malloc + 24
   _main + 24           [x.f:5]
       1000022C
1546-520 Memory contents:  61626364                                [abcd
1546-515 -----------------------------------------------------------------
1546-518 Address: 0x2000DE10      Size: 0x00000004 (4)
    _int_debug_umalloc + 32C
        _debug_umalloc + 44
   _dbg_umalloc + 18
```

```
        malloc + 24
   _main + 64          [x.f:11]
      1000022C
1546-520 Memory contents:  00000001                              [....
1546-515 ----------------------------------------------------------------------
1546-517                  END OF DUMP OF ALLOCATED MEMORY BLOCKS
1546-515 ----------------------------------------------------------------------
1546-515 ----------------------------------------------------------------------
1546-516                  START OF DELTA DUMP OF ALLOCATED MEMORY BLOCKS
1546-515 ----------------------------------------------------------------------
1546-518 Address: 0x2000DE30      Size: 0x00000004 (4)
    _int_debug_umalloc + 32C
        _debug_umalloc + 44
   _dbg_umalloc + 18
        malloc + 24
   _main + 8C          [x.f:16]
      1000022C
1546-520 Memory contents:  00000002                              [....
1546-515 ----------------------------------------------------------------------
1546-517                  END OF DELTA DUMP OF ALLOCATED MEMORY BLOCKS
1546-515 ----------------------------------------------------------------------
```

Example 2: Invalid write

```
   pointer (p,a)                  ! 1
   integer a                      ! 2
          ! 3
   p = malloc(%val(4))            ! 4
   a = 1                          ! 5
   p = p + 4                      ! 6
   a = 2                          ! 7
          ! 8
   call _heap_check()              ! 9
          ! 10
   end                            ! 11
```

Output:

```
1546-503 End of allocated object 0x20000BD0 was overwritten at 0x20000BD4.
1546-514 The first eight bytes of the object (in hex) are: 0000000100000002.
    _int_debug_umalloc + 32C
        _debug_umalloc + 44
   _dbg_umalloc + 18
  _umalloc_init + 30
        malloc + 24
   _main + 24          [x.f:4]
      1000022C
1546-522 Traceback:
    0xD09D1C94 = _uheap_check_init + 0x24
    0xD09D18C0 = heap_check + 0x28
    0x100002C8 = _main + 0x5C
IOT/Abort trap(coredump)
```

Understanding XL Fortran Compiler Listings

Diagnostic information is placed in the output listing produced by the **-qlist**, **-qsource**, **-qxref**, **-qattr**, **-qreport**, and **-qlistopt** compiler options. The **-S** option generates an assembler listing in a separate file.

To locate the cause of a problem with the help of a listing, you can refer to the following:

- The source section (to see any compilation errors in the context of the source program)
- The attribute and cross-reference section (to find data objects that are misnamed or used without being declared or to find mismatched parameters)
- The transformation and object sections (to see if the generated code is similar to what you expect)

A heading identifies each major section of the listing. A string of greater than symbols precede the section heading so that you can easily locate its beginning:

```
>>>>> section name
```

You can select which sections appear in the listing by specifying compiler options.

Related Information: See "Options That Control Listings and Messages" on page 77.

Header Section

The listing file has a header section that contains the following items:

- A compiler identifier that consists of the following:
  - Compiler name
  - Version number
  - Release number
  - Modification number
  - Fix number
- Source file name
- Date of compilation
- Time of compilation

The header section is always present in a listing; it is the first line and appears only once. The following sections are repeated for each compilation unit when more than one compilation unit is present.

Options Section

The options section is always present in a listing. There is a separate section for each compilation unit. It indicates the specified options that are in effect for the compilation unit. This information is useful when you have conflicting options. If you specify the **-qlistopt** compiler option, this section lists the settings for all options.

Source Section

The source section contains the input source lines with a line number and, optionally, a file number. The file number indicates the source file (or include file) from which the source line originated. All main file source lines (those that are not from an include file) do not have the file number printed. Each include file has a file number associated with it, and source lines from include files have that file number printed. The file number appears on the left, the line number appears to its right, and the text of the source line is to the right of the line number. XL Fortran numbers lines relative to each file. The source lines and the numbers that are associated with them appear only if the **-qsource** compiler option is in effect. You can selectively print parts of the source by using the **@PROCESS** directives **SOURCE** and **NOSOURCE** throughout the program.

Error Messages

If the **-qsource** option is in effect, the error messages are interspersed with the source listing. The error messages that are generated during the compilation process contain the following:

- The source line
- A line of indicators that point to the columns that are in error
- The error message, which consists of the following:
  - The 4-digit component number
  - The number of the error message
  - The severity level of the message
  - The text that describes the error

For example:

```
          2 |         equivalence (i,j,i)
     ............................a.
a - 1514-092: (E) Same name appears more than once in an
equivalence group.
```

If the **-qnosource** option is in effect, the error messages are all that appear in the source section, and an error message contains:

- The file name in quotation marks
- The line number and column position of the error
- The error message, which consists of the following:
  - The 4-digit component number
  - The number of the error message
  - The severity level of the message
  - The text that describes the error

For example:

```
"doc.f", line 6.11: 1513-039 (S) Number of arguments is not
permitted for INTRINSIC function abs.
```

Transformation Report Section

If the **-qreport** option is in effect, a transformation report listing shows how XL Fortran optimized the program. This section displays pseudo-Fortran code that corresponds to the original source code, so that you can see parallelization and loop transformations that the **-qhot** and/or **-qsmp** options have generated.

Sample Report

The following report was created for the program **t.f** using the

```
xlf -qhot -qreport t.f
```

command.

Program t.f:

```
        integer a(100, 100)
        integer i,j

        do i = 1 , 100
          do j = 1, 100
            a(i,j) = j
          end do
        end do
        end
```

Transformation Report:

```
>>>>> SOURCE SECTION <<<<<
** _main   === End of Compilation 1 ===

>>>>> LOOP TRANSFORMATION SECTION <<<<<



                PROGRAM _main ()
     4|           IF (.FALSE.) GOTO lab_9
                  @LoopIV0 = 0
       Id=1       DO @LoopIV0 = @LoopIV0, 99
     5|             IF (.FALSE.) GOTO lab_11
                    @LoopIV1 = 0
       Id=2         DO @LoopIV1 = @LoopIV1, 99
                      ! DIR_INDEPENDENT loopId = 0
     6|               a((@LoopIV1 + 1),(@LoopIV0 + 1)) = (@LoopIV0 + 1)
     7|             ENDDO
                    lab_11
     8|           ENDDO
                  lab_9
     9|         END PROGRAM _main


Source        Source        Loop Id       Action / Information
File          Line

----------    ----------    ----------    -------------------------------------
         0             4             1    Loop interchanging applied to loop nest.

>>>>> FILE TABLE SECTION <<<<<
```

Attribute and Cross-Reference Section

This section provides information about the entities that are used in the compilation unit. It is present if the **-qxref** or **-qattr** compiler option is in effect. Depending on the options in effect, this section contains all or part of the following information about the entities that are used in the compilation unit:

- Names of the entities
- Attributes of the entities (if **-qattr** is in effect). Attribute information may include any or all of the following details:
  - The type
  - The class of the name
  - The relative address of the name
  - Alignment
  - Dimensions
  - For an array, whether it is allocatable
  - Whether it is a pointer, target, or integer pointer
  - Whether it is a parameter
  - Whether it is volatile
  - For a dummy argument, its intent, whether it is value, and whether it is optional
  - Private, public, protected, module
- Coordinates to indicate where you have defined, referenced, or modified the entities. If you declared the entity, the coordinates are marked with a `$`. If you initialized the entity, the coordinates are marked with a `*`. If you both declared and initialized the entity at the same place, the coordinates are marked with a `&`. If the entity is set, the coordinates are marked with a `@`. If the entity is referenced, the coordinates are not marked.

Class is one of the following:

- Automatic
- BSS (uninitialized static internal)
- Common
- Common block
- Construct name
- Controlled (for an allocatable object)
- Controlled automatic (for an automatic object)
- Defined assignment
- Defined operator
- Derived type definition
- Entry
- External subprogram
- Function
- Generic name
- Internal subprogram
- Intrinsic
- Module
- Module function
- Module subroutine
- Namelist
- Pointee
- Private component
- Program
- Reference parameter
- Renames
- Static
- Subroutine

- Use associated
- Value parameter

Type is one of the following:
- Byte
- Character
- Complex
- Derived type
- Integer
- Logical
- Real

If you specify the **full** suboption with **-qxref** or **-qattr**, XL Fortran reports all entities in the compilation unit. If you do not specify this suboption, only the entities you actually use appear.

Object Section

XL Fortran produces this section only when the **-qlist** compiler option is in effect. It contains the object code listing, which shows the source line number, the instruction offset in hexadecimal notation, the assembler mnemonic of the instruction, and the hexadecimal value of the instruction. On the right side, it also shows the cycle time of the instruction and the intermediate language of the compiler. Finally, the total cycle time (straight-line execution time) and the total number of machine instructions that are produced are displayed. There is a separate section for each compilation unit.

File Table Section

This section contains a table that shows the file number and file name for each main source file and include file used. It also lists the line number of the main source file at which the include file is referenced. This section is always present.

Compilation Unit Epilogue Section

This is the last section of the listing for each compilation unit. It contains the diagnostics summary and indicates whether the unit was compiled successfully. This section is not present in the listing if the file contains only one compilation unit.

Compilation Epilogue Section

The compilation epilogue section occurs only once at the end of the listing. At completion of the compilation, XL Fortran presents a summary of the compilation: number of source records that were read, compilation start time, compilation end time, total compilation time, total CPU time, and virtual CPU time. This section is always present in a listing.

Related Information: Sample programs are shown in Appendix A, “Sample Fortran Programs,” on page 405.

Fortran-Related AIX Commands

You can use some AIX commands with XL Fortran files to perform several tasks, as outlined in this section.

Related Information: See the *AIX Commands Reference* for full details on these commands.

Working with Object-Code Archives (ar)

The **ar** command performs operations on libraries of object files that are used during linking. You might use this command to create a library of support routines that can be linked into many different programs, as follows:

```
ar -q ~/mylibs/graphics.a raytrace.o shade.o illuminate.o
xlf95 spheres.f -L~/mylibs/ -lgraphics
```

Printing Output Files with Fortran ASA Carriage Controls (asa)

The **asa** command translates the output of Fortran programs that use the historical Fortran convention for ASA carriage-control characters.

The translated output is suitable for the **qprt** command:

```
generate_output | asa | qprt
```

The **fpr** command is the same as the **asa** command.

For a list of supported carriage-control characters, see *Formatted Records* in the *XL Fortran Enterprise Edition for AIX Language Reference*.

Splitting Subprograms into Individual Files (fsplit)

The **fsplit** command splits the specified Fortran source program files into several files. You should only use **fsplit** with FORTRAN 77 programs.

```
$ cat fsplit.f
        subroutine sub1
                print *,'Hello world'
        end

        subroutine sub2
                print *,'Goodbye'
        end

        program main
                call sub1
                call sub2
        end
$ fsplit fsplit.f
sub1.f
sub2.f
main.f
```

Automating Large, Complex Compilations (make)

The **make** command allows you to specify rules (sets of commands and options) to use for processing different types of files. By keeping track of which files are out-of-date and need to be recompiled, it can automate some or all aspects of the compilation process.

If you use **make** with XL Fortran, you may not want an object or an executable file created when the compiler encounters errors. The default setting of **-qhalt=s** prevents the compiler from generating the object file if it finds problems that it cannot correct.

Important: If you make any changes to the default configuration file and then move or copy your makefiles to another system, you also need to copy the changed configuration file.

Run-Time Profiling (prof, gprof)

The **prof** and **gprof** commands provide different levels of run-time profile reports, which you can examine to find performance bottlenecks and identify the subprograms that are called most or least often. This information can help you decide where to concentrate performance-tuning efforts.

See "-p Option" on page 118 for information on compiling for profiling and an example of the sequence of commands.

The "-qipa Option" on page 182 allows you to feed profiling information back into subsequent compilations to enhance optimization.

Translating Programs into RATFOR (struct)

The **struct** command translates a FORTRAN 77 source program into a RATFOR program:

```
struct fortran.f >ratfor.f
```

Displaying Information inside Binary Files (what)

The **what** command reads information encoded into some binary files, as follows:
- Information about the compiler version is encoded in **/usr/lpp/xlf/bin/xlfentry**.
- 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.

Porting Programs to XL Fortran

XL Fortran provides many features intended to make it easier to take programs that were originally written for other computer systems or compilers and recompile them with XL Fortran.

Outline of the Porting Process

The process for porting a typical program looks like this:

1. Identify any nonportable language extensions or subroutines that you used in the original program. Check to see which of these XL Fortran supports:
   - Language extensions are identified in the *XL Fortran Enterprise Edition for AIX Language Reference*.
   - Some extensions require you to specify an XL Fortran compiler option; you can find these options listed in Table 8 on page 80.
2. For any nonportable features that XL Fortran does not support, modify the source files to remove or work around them.
3. Do the same for any implementation-dependent features. For example, if your program relies on the representation of floating-point values or uses system-specific file names, you may need to change it.
4. Compile the program with XL Fortran. If any compilation problems occur, fix them and recompile and fix any additional errors until the program compiles successfully.
5. Run the XLF-compiled program and compare the output with the output from the other system. If the results are substantially different, there are probably still some implementation-specific features that need to be changed. If the results are only marginally different (for example, if XL Fortran produces a different number of digits of precision or a number differs in the last decimal place), decide whether the difference is significant enough to investigate further. You may be able to fix these differences.

Before porting programs to XL Fortran, read the tips in the following sections so that you know in advance what compatibility features XL Fortran offers.

Maintaining FORTRAN 77 Source and Object Code

You can recompile existing FORTRAN 77 programs from XL Fortran Version 2 with XL Fortran Version 9.1.

You can link existing FORTRAN 77 object code from XL Fortran Versions 1 to 8 into programs generated by XL Fortran Version 9.1. See "Linking New Objects with Existing Ones" on page 45 for details.

Portability of Directives

XL Fortran supports many directives available with other Fortran products. This ensures easy portability between products. If your code contains *trigger_constants* other than the defaults in XL Fortran, you can use the **-qdirective** compiler option to specify them. For instance, if you are porting CRAY code contained in a file `xx.f`, you would use the following command to add the CRAY *trigger_constant*:

```
xlf95 xx.f -qdirective=mic\$
```

For fixed source form code, in addition to the **!** value for the *trigger_head* portion of the directive, XL Fortran also supports the *trigger_head* values **C**, **c**, and *****.

For more information, see "-qdirective Option" on page 148.

XL Fortran supports a number of programming terms as synonyms to ease the effort of porting code from other Fortran products. Those terms that are supported are dependent on context, as indicated in the following tables:

Table 30. PARALLEL DO Clauses and Their XL Fortran Synonyms

PARALLEL DO Clause	**XL Fortran Synonym**
LASTLOCAL	LASTPRIVATE
LOCAL	PRIVATE
MP_SCHEDTYPE *and* CHUNK	SCHEDULE
SAVELAST	LASTPRIVATE
SHARE	SHARED
NEW	PRIVATE

Table 31. PARALLEL DO Scheduling Types and Their XL Fortran Synonyms

Scheduling Type	**XL Fortran Synonym**
GSS	GUIDED
INTERLEAVE	STATIC(1)
INTERLEAVED	STATIC(1)
INTERLEAVE(n)	STATIC(n)
INTERLEAVED(n)	STATIC(n)
SIMPLE	STATIC

Table 32. PARALLEL SECTIONS Clauses and Their XL Fortran Synonyms

PARALLEL SECTIONS Clause	**XL Fortran Synonym**
LOCAL	PRIVATE
SHARE	SHARED
NEW	PRIVATE

NEW

Use the **NEW** directive to specify which variables should be local in a **PARALLEL DO** loop or a **PARALLEL SECTIONS** construct. This directive performs the same function as the **PRIVATE** clause of the **PARALLEL DO** directive and **PARALLEL SECTIONS** directive.

Background Information

The **NEW** directive only takes effect if you specify the **-qsmp** compiler option.

Syntax

```
►►──NEW──named_variable_list──────────────────────────────────────────────►◄
```

The **NEW** directive must immediately follow either a **PARALLEL DO** directive or a **PARALLEL SECTIONS** directive.

If you specify the **NEW** directive, you must specify the corresponding **PARALLEL DO** or **PARALLEL SECTIONS** directive with no clauses.

If the **NEW** directive follows the **PARALLEL DO** directive, the first noncomment line (not including other directives) following the **NEW** directive must be a **DO** loop. This line cannot be an infinite **DO** or **DO WHILE** loop.

A variable name in the *named_variable_list* of the **NEW** directive has the same restrictions as a variable name appearing in the **PRIVATE** clause of the **PARALLEL DO** directive or a **PRIVATE** clause of the **PARALLEL SECTIONS** directive. See the sections on the **PARALLEL DO** directive and the **PARALLEL SECTIONS** construct in the *XL Fortran Enterprise Edition for AIX Language Reference*.

Examples

```
INTEGER A(10), C(10)
REAL B(10)
INTEGER FUNC(100)
!SMP$ PARALLEL DO
!SMP$ NEW I, TMP
        DO I = 1, 10
          TMP = A(I) + COS(B(I))
          C(I) = TMP + FUNC(I)
        END DO
```

Common Industry Extensions That XL Fortran Supports

XL Fortran allows many of the same FORTRAN 77 extensions as other popular compilers, including:

Extension	**Refer to** *XL Fortran Enterprise Edition for AIX Language Reference* **Section(s)**
Typeless constants	*Typeless Literal Constants*
**len* length specifiers for types	*The Data Types*
BYTE data type	**BYTE**
Long variable names	*Names*
Lower case	*Names*
Mixing integers and logicals (with **-qintlog** option)	*Evaluation of Expressions*
Character-count **Q** edit descriptor (with **-qqcount** option)	*Q (Character Count) Editing*
Intrinsics for counting set bits in registers and determining data-object parity	**POPCNT**, **POPPAR**
64-bit data types (**INTEGER(8)**, **REAL(8)**, **COMPLEX(8)**, and **LOGICAL(8)**), including support for default 64-bit types (with **-qintsize** and **-qrealsize** options)	*Integer Real Complex Logical*
Integer **POINTER**s, similar to those supported by CRAY and Sun compilers. (XL Fortran integer pointer arithmetic uses increments of one byte, while the increment on CRAY computers is eight bytes. You may need to multiply pointer increments and decrements by eight to make programs ported from CRAY computers work properly.)	**POINTER(integer)**
Conditional vector merge (CVMGx) intrinsic functions	*CVMGx (TSOURCE, FSOURCE, MASK)*
Date and time service and utility functions (rtc, irtc, jdate, clock_, timef, and date)	*Service and Utility Procedures*
STRUCTURE, **UNION**, and **MAP** constructs	*Structure Components*, *Union and Map*

Mixing Data Types in Statements

The **-qctyplss** option lets you use character constant expressions in the same places that you use typeless constants. The **-qintlog** option lets you use integer expressions where you can use logicals, and vice versa. A kind type parameter must not be replaced with a logical constant even if **-qintlog** is on, nor by a character constant even if **-qctyplss** is on, nor can it be a typeless constant.

Date and Time Routines

Date and time routines, such as **dtime**, **etime**, and **jdate**, are accessible as Fortran subroutines.

Other libc Routines

A number of other popular routines from the **libc** library, such as **flush**, **getenv**, and **system**, are also accessible as Fortran subroutines.

Changing the Default Sizes of Data Types

For porting from machines with larger or smaller word sizes, the **-qintsize** option lets you specify the default size for integers and logicals. The **-qrealsize** option lets you specify the default size for reals and complex components.

Name Conflicts between Your Procedures and XL Fortran Intrinsic Procedures

If you have procedures with the same names as any XL Fortran intrinsic procedures, the program calls the intrinsic procedure. (This situation is more likely with the addition of the many new Fortran 90 and Fortran 95 intrinsic procedures.)

If you still want to call your procedure, add explicit interfaces, **EXTERNAL** statements, or **PROCEDURE** statements for any procedures with conflicting names, or use the **-qextern** option when compiling.

Reproducing Results from Other Systems

XL Fortran provides settings through the **-qfloat** option that help make floating-point results consistent with those from other IEEE systems; this subject is discussed in "Duplicating the Floating-Point Results of Other Systems" on page 295.

Finding Nonstandard Extensions

XL Fortran supports a number of extensions to various language standards. Many of these extensions are so common that you need to keep in mind, when you port programs to other systems, that not all compilers have them. To find such extensions in your XL Fortran programs before beginning a porting effort, use the **-qlanglvl** option:

```
$ # -qnoobject stops the compiler after parsing all the source,
$ # giving a fast way to check for errors.
$ # Look for anything above the base F77 standard.
$ xlf  -qnoobject -qlanglvl=77std f77prog.f
  ...
$ # Look for anything above the F90 standard.
$ xlf90 -qnoobject -qlanglvl=90std use_in_2000.f
  ...
$ # Look for anything above the F95 standard.
$ xlf95 -qnoobject -qlanglvl=95std use_in_2000.f
  ...
```

Related Information: See "-qlanglvl Option" on page 189 and "-qport Option" on page 217.

Answers to Frequently Asked Questions

Here are the answers to some questions that are often asked by users of XL Fortran. Many of these questions are answered elsewhere in the XL Fortran documentation, but they are also collected here for your convenience.

Finding the Date and Time

There are some common date and time subprograms that you may be familiar with from programming in FORTRAN 77 on other systems. XL Fortran provides equivalents to many of these subprograms. Some of the names have trailing underscores to avoid conflicts with C library functions of the same name.

The XL Fortran subprograms that deal with the date and time are:

alarm_
clock_
ctime_
date
dtime_
etime_
fdate_
gmtime_
idate_
irtc
itime_
jdate
ltime_
rtc
sleep_
time_
timef
usleep_

See the section on *Service and Utility Procedures* in the *XL Fortran Enterprise Edition for AIX Language Reference* for details about these subprograms.

For portable Fortran 90 and Fortran 95 programming, you can also use the **CPU_TIME**, **DATE_AND_TIME**, and **SYSTEM_CLOCK** intrinsic subroutines.

Efficient Static Linking

"Dynamic and Static Linking" on page 46 discusses the respective strengths and weaknesses of static and dynamic linking. A technique for static linking that requires relatively little disk space is to link any XL Fortran libraries statically but leave references to other system libraries dynamic. This example statically links just the XL Fortran library:

```
# Build a temporary object:
   ld -r -o libtmp.o -bnso -lxlf90
# Build the application with this object on the command line:
   xlf95 -o appl appl1.o appl2.o libtmp.o
```


Appendix A. Sample Fortran Programs

The following programs are provided as coding examples for XL Fortran. A number of these samples illustrate various aspects of SMP programming that may be new to many users. If you are new to SMP programming, you should examine these samples to gain a better understanding of the SMP coding style. Every attempt has been made to internally document key areas of the source to assist you in this effort.

You can compile and execute the first program to verify that the compiler is installed correctly and your user ID is set up to execute Fortran programs.

Example 1 - XL Fortran Source File

```
      PROGRAM CALCULATE
!
! Program to calculate the sum of up to n values of x**3
! where negative values are ignored.
!
      IMPLICIT NONE
      INTEGER I,N
      REAL SUM,X,Y
      READ(*,*) N
      SUM=0
      DO I=1,N
         READ(*,*) X
         IF (X.GE.0) THEN
            Y=X**3
            SUM=SUM+Y
         END IF
      END DO
      WRITE(*,*) 'This is the sum of the positive cubes:',SUM
      END
```

Execution Results

Here is what happens when you run the program:

```
$ a.out
5
37
22
-4
19
6
 This is the sum of the positive cubes:  68376.00000
```

Example 2 - Valid C Routine Source File

```
/*
 * ********************************************************************
 * This is a main function that creates threads to execute the Fortran
 * test subroutines.
 * ********************************************************************
 */
#include <pthread.h>
#include <stdio.h>
#include <errno.h>

extern char *sys_errlist[];
extern char *optarg;
extern int optind;

static char *prog_name;

#define MAX_NUM_THREADS 100

void *f_mt_exec(void *);
void f_pre_mt_exec(void);
void f_post_mt_exec(int *);

void
usage(void)
{
    fprintf(stderr, "Usage: %s -t number_of_threads.\n", prog_name);
    exit(-1);
}

main(int argc, char *argv[])
{
    int i, c, rc;
    int num_of_threads, n[MAX_NUM_THREADS];
    char *num_of_threads_p;
    pthread_attr_t attr;
    pthread_t tid[MAX_NUM_THREADS];

    prog_name = argv[0];
    while ((c = getopt(argc, argv, "t")) != EOF)
    {
        switch (c)
        {
        case 't':
            break;

        default:
            usage();
            break;
        }
    }
    argc -= optind;
    argv += optind;
    if (argc < 1)
    {
        usage();
    }

    num_of_threads_p = argv[0];
    if ((num_of_threads = atoi(num_of_threads_p)) == 0)
    {
        fprintf(stderr,
         "%s: Invalid number of threads to be created <%s>\n", prog_name,
                num_of_threads_p);
        exit(1);
```

```
    }
    else if (num_of_threads > MAX_NUM_THREADS)
    {
        fprintf(stderr,
                "%s: Cannot create more than 100 threads.\n", prog_name);
        exit(1);
    }
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_UNDETACHED);

    /* ****************************************************************
     * Execute the Fortran subroutine that prepares for multi-threaded
     * execution.
     * ****************************************************************
     */
    f_pre_mt_exec();

    for (i = 0; i < num_of_threads; i++)
    {
        n[i] = i;
        rc = pthread_create(&tid[i], &attr, f_mt_exec, (void *)&n[i]);
        if (rc != 0)
        {
            fprintf(stderr, "Failed to create thread %d.\n", i);
            fprintf(stderr, "Error is %s\n", sys_errlist[rc]);
            exit(1);
        }
    }
    /* The attribute is no longer needed after threads are created. */
    pthread_attr_destroy(&attr);
    for (i = 0; i < num_of_threads; i++)
    {
        rc = pthread_join(tid[i], NULL);
        if (rc != 0)
        {
            fprintf(stderr, "Failed to join thread %d. \n", i);
            fprintf(stderr, "Error is %s\n", sys_errlist[rc]);
        }
    }
    /*
     * Execute the Fortran subroutine that does the check after
     * multi-threaded execution.
     */
    f_post_mt_exec(&num_of_threads);

    exit(0);
}

! ***********************************************************************
! This test case tests the writing list-directed to a single external
! file by many threads.
! ***********************************************************************

        subroutine f_pre_mt_exec()
        integer array(1000)
        common /x/ array

        do i = 1, 1000
          array(i) = i
        end do

        open(10, file="fun10.out", form="formatted", status="replace")
        end

        subroutine f_post_mt_exec(number_of_threads)
        integer array(1000), array1(1000)
        common /x/ array
```

```
      close(10)
      open(10, file="fun10.out", form="formatted")
      do j = 1, number_of_threads
        read(10, *) array1

        do i = 1, 1000
          if (array1(i) /= array(i)) then
            print *, "Result is wrong."
            stop
          endif
        end do
      end do
      close(10, status="delete")
      print *, "Normal ending."
      end

      subroutine f_mt_exec(thread_number)
      integer thread_number
      integer array(1000)
      common /x/ array

      write(10, *) array
      end
```

Example 3 - Valid Fortran SMP Source File

```
!*****************************************************************
!* This example uses a PARALLEL construct and a DO construct     *
!* to calculate the value of pi.                                 *
!*****************************************************************
      program compute_pi
      integer n, i
      real*8 w, x, pi, f, a
      f(a) = 4.d0 /(1.d0 + a*a)  !! function to integrate

      pi = 0.0d0
!$OMP PARALLEL private(x, w, n), shared(pi)
      n = 10000                     !! number of intervals
      w = 1.0d0/n                   !! calculate the interval size
!$OMP DO reduction(+: pi)
      do i = 1, n
         x = w * (i - 0.5d0)
         pi = pi + f(x)
      enddo
!$OMP END DO
!$OMP END PARALLEL
      print *, "Computed pi = ", pi
     end
```

Example 4 - Invalid Fortran SMP Source File

```
!*****************************************************************
!* In this example, fort_sub is invoked by multiple threads.     *
!*                                                               *
!* This example is not valid because                             *
!*  fort_sub and another_sub both declare /block/ to be          *
!*  THREADLOCAL. They intend to share the common block, but      *
!*  they are executed via different threads.                     *
!*                                                               *
!* To "fix" this problem, one of the following approaches can    *
!* be taken:                                                     *
!*  (1) The code for another_sub should be brought into the loop.*
!*  (2) "j" should be passed as an argument to another_sub, and  *
!*      the declaration for /block/ should be removed from       *
!*      another_sub.                                             *
```

```
!*  (3) The loop should be marked as "do not parallelize" by     *
!*      using the directive "!SMP$ PARALLEL DO  IF(.FALSE.)".    *
!*****************************************************************

subroutine fort_sub()

  common /block/ j
  integer :: j
  !IBM* THREADLOCAL /block/         ! Each thread executing fort_sub
                                    ! obtains its own copy of /block/.
  integer a(10)

  ...
  !IBM* INDEPENDENT
  do index = 1,10
    call another_sub(a(i))
  enddo
  ...

end subroutine fort_sub

subroutine another_sub(aa)          ! Multiple threads are used to
  integer aa                        ! execute another_sub.
  common /block/ j                  ! Each thread obtains a new copy
  integer :: j                      ! of the common block /block/.
  !IBM* THREADLOCAL /block/

  aa = j                            ! The value of "j" is undefined.
end subroutine another_sub
```

Programming Examples Using the Pthreads Library Module

```
!******************************************************************
!* Example 5 : Create a thread with Round_Robin scheduling policy.*
!* For simplicity, we do not show any codes for error checking,   *
!* which would be necessary in a real program.                    *
!******************************************************************
        use, intrinsic::f_pthread
        integer(4) ret_val
        type(f_pthread_attr_t) attr
        type(f_pthread_t)      thr

        ret_val = f_pthread_attr_init(attr)
        ret_val = f_pthread_attr_setschedpolicy(attr, SCHED_RR)
        ret_val = f_pthread_attr_setinheritsched(attr, PTHREAD_EXPLICIT_SCHED)
        ret_val = f_pthread_create(thr, attr, FLAG_DEFAULT, ent, integer_arg)
        ret_val = f_pthread_attr_destroy(attr)
        ......
```

Before you can manipulate a pthread attribute object, you need to create and initialize it. The appropriate interfaces must be called to manipulate the attribute objects. A call to **f_pthread_attr_setschedpolicy** sets the scheduling policy attribute to Round_Robin. Note that this does not affect newly created threads that inherit the scheduling property from the creating thread. For these threads, we explicitly call **f_pthread_attr_setinheritsched** to override the default inheritance attribute. The rest of the code is self-explanatory.

```
!*****************************************************************
!* Example 6 : Thread safety                                     *
!* In this example, we show that thread safety can be achieved   *
!* by using the push-pop cleanup stack for each thread. We       *
!* assume that the thread is in deferred cancellability-enabled  *
!* state.  This means that any thread-cancel requests will be    *
!* put on hold until a cancellation point is encountered.       *
!* Note that f_pthread_cond_wait provides a                      *
!* cancellation point.                                           *
```

```
!***************************************************************
       use, intrinsic::f_pthread
       integer(4) ret_val
       type(f_pthread_mutex_t) mutex
       type(f_pthread_cond_t) cond
       pointer(p, byte)
       ! Initialize mutex and condition variables before using them.
       ! For global variables this should be done in a module, so that they
       ! can be used by all threads. If they are local, other threads
       ! will not see them. Furthermore, they must be managed carefully
       ! (for example, destroy them before returning, to avoid dangling and
       ! undefined objects).
       mutex = PTHREAD_MUTEX_INITIALIZER
       cond  = PTHREAD_COND_INITIALIZER

       ......
       ! Doing something

       ......

       ! This thread needs to allocate some memory area used to
       ! synchronize with other threads. However, when it waits on a
       ! condition variable, this thread may be canceled by another
       ! thread. The allocated memory may be lost if no measures are
       ! taken in advance. This will cause memory leakage.

       ret_val = f_pthread_mutex_lock(mutex)
       p = malloc(%val(4096))

       ! Check condition. If it is not true, wait for it.
       ! This should be a loop.

       ! Since memory has been allocated, cleanup must be registered
       ! for safety during condition waiting.

       ret_val = f_pthread_cleanup_push(mycleanup, FLAG_DEFAULT, p)
       ret_val = f_pthread_cond_wait(cond, mutex)

       ! If this thread returns from condition waiting, the cleanup
       ! should be de-registered.

       call f_pthread_cleanup_pop(0)      ! not execute
       ret_val = f_pthread_mutex_unlock(mutex)

       ! This thread will take care of p for the rest of its life.
       ......


       ! mycleanup looks like:

       subroutine mycleanup(passed_in)
           pointer(passed_in, byte)
           external free

           call free(%val(passed_in))
       end subroutine mycleanup
```

Appendix B. XL Fortran Technical Information

This section contains details about XL Fortran that advanced programmers may need to diagnose unusual problems, run the compiler in a specialized environment, or do other things that a casual programmer is rarely concerned with.

The Compiler Phases

The typical compiler invocation command executes some or all of the following programs in sequence. For link-time optimizations, some of the phases will be executed more than once during a compilation. As each program runs, the results are sent to the next step in the sequence.

1. A preprocessor
2. The compiler, which consists of the following phases:
   a. Front-end parsing and semantic analysis
   b. Loop transformations
   c. Interprocedural analysis
   d. Optimization
   e. Register allocation
   f. Final assembly
3. The assembler (for any **.s** files)
4. The linker **ld**

External Names in theXL FortranShared Libraries

The run-time libraries included in the XL Fortran Run-Time Environment are AIX shared libraries, which are processed by the linker to resolve all references to external names. To minimize naming conflicts between user-defined names and the names that are defined in the run-time libraries, the names of input/output routines in the run-time libraries are prefixed with an underscore(_), or _xl.

The XL Fortran Run-Time Environment

Object code that the XL Fortran compiler produces often invokes compiler-supplied subprograms at run time to handle certain complex tasks. These subprograms are collected into several libraries.

The function of the XL Fortran Run-Time Environment may be divided into these main categories:
- Support for Fortran I/O operations
- Mathematical calculation
- Operating-system services
- Support for SMP parallelization

The XL Fortran Run-Time Environment also produces run-time diagnostic messages in the national language appropriate for your system. Unless you bind statically, you cannot run object code produced by the XL Fortran compiler without the XL Fortran Run-Time Environment.

The XL Fortran Run-Time Environment is upward-compatible. Programs that are compiled with a given level of the run-time environment and a given level of the operating system require the same or higher levels of both the run-time environment and the operating system to run.

External Names in the Run-Time Environment

Run-time subprograms are collected into libraries. By default, the compiler invocation command also invokes the linker and gives it the names of the libraries that contain run-time subprograms called by Fortran object code.

The names of these run-time subprograms are external symbols. When object code that is produced by the XL Fortran compiler calls a run-time subprogram, the **.o** object code file contains an external symbol reference to the name of the subprogram. A library contains an external symbol definition for the subprogram. The linker resolves the run-time subprogram call with the subprogram definition.

You should avoid using names in your XL Fortran program that conflict with names of run-time subprograms. Conflict can arise under two conditions:
- The name of a subroutine, function, or common block that is defined in a Fortran program has the same name as a library subprogram.
- The Fortran program calls a subroutine or function with the same name as a library subprogram but does not supply a definition for the called subroutine or function.

Technical Details of the -qfloat=hsflt Option

The **-qfloat=hsflt** option is unsafe for optimized programs that compute floating-point values that are outside the range of representation of single precision, not just outside the range of the result type. The range of representation includes both the precision and the exponent range.

Even when you follow the rules that are stated in the preceding paragraph and in “-qfloat Option” on page 163, programs that are sensitive to precision differences might not produce expected results. Because **-qfloat=hsflt** is not compliant with IEEE in a number of ways, programs may not run correctly. If a program gives unexpected, incorrect, or unacceptable results when compiled with this option, use **-qfloat=hssngl** instead.

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.

The speedup from this option is primarily for POWER and POWER2 machines. We recommend that it not be used for programs targeted (through the **-qarch** option) for PowerPC machines.

Implementation Details for -qautodbl Promotion and Padding

The following sections provide additional details about how the **-qautodbl** option works, to allow you to predict what happens during promotion and padding.

Terminology

The *storage relationship* between two data objects determines the relative starting addresses and the relative sizes of the objects. The **-qautodbl** option tries to preserve this relationship as much as possible.

Data objects can also have a *value relationship*, which determines how changes to one object affect another. For example, a program might store a value into one variable, and then read the value through a different storage-associated variable. With **-qautodbl** in effect, the representation of one or both variables might be different, so the value relationship is not always preserved.

An object that is affected by this option may be:

- *Promoted*, meaning that it is converted to a higher-precision data type. Usually, the resulting object is twice as large as it would be by default. Promotion applies to constants, variables, derived-type components, arrays, and functions (which include intrinsic functions) of the appropriate types.

  Note: **BYTE**, **INTEGER**, **LOGICAL**, and **CHARACTER** objects are never promoted.
- *Padded*, meaning that the object keeps its original type but is followed by undefined storage space. Padding applies to **BYTE**, **INTEGER**, **LOGICAL**, and nonpromoted **REAL** and **COMPLEX** objects that may share storage space with promoted items. For safety, **POINTER**s, **TARGET**s, actual and dummy arguments, members of **COMMON** blocks, structures, pointee arrays, and pointee **COMPLEX** objects are always padded appropriately depending on the **-qautodbl** suboption. This is true whether or not they share storage with promoted objects.

  Space added for padding ensures that the storage-sharing relationship that existed before conversion is maintained. For example, if array elements **I(20)** and **R(10)** start at the same address by default and if the elements of **R** are promoted and become twice as large, the elements of **I** are padded so that **I(20)** and **R(10)** still start at the same address.

  Except for unformatted I/O statements, which read and write any padding that is present within structures, I/O statements do not process padding.

  Note: The compiler does not pad **CHARACTER** objects.

Examples of Storage Relationships for -qautodbl Suboptions

The examples in this section illustrate storage-sharing relationships between the following types of entities:

- **REAL(4)**
- **REAL(8)**
- **REAL(16)**
- **COMPLEX(4)**
- **COMPLEX(8)**
- **COMPLEX(16)**
- **INTEGER(8)**
- **INTEGER(4)**
- **CHARACTER(16)**.

Note: In the diagrams, solid lines represent the actual data, and dashed lines represent padding.

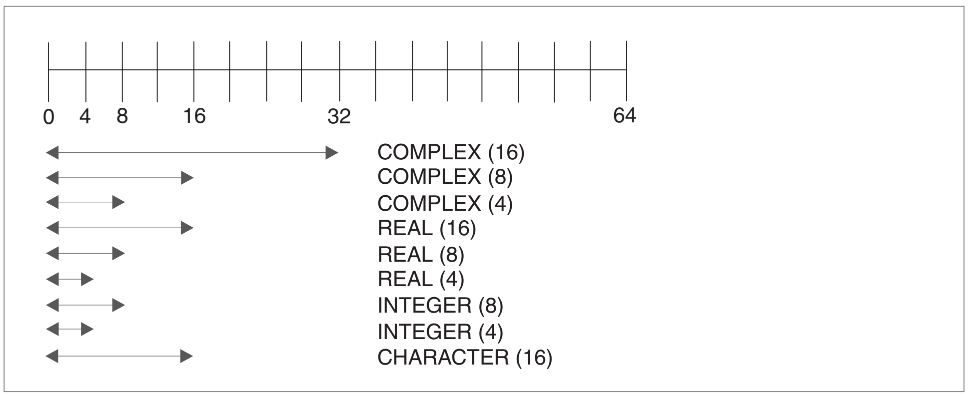


Figure 6. Storage Relationships without the -qautodbl Option

The figure above illustrates the default storage-sharing relationship of the compiler.

```
@process autodbl(none)
      block data
      complex(4) x8      /(1.123456789e0,2.123456789e0)/
      real(16) r16(2)    /1.123q0,2.123q0/
      integer(8) i8(2)   /1000,2000/
      character*5 c(2)  /"abcde","12345"/
      common /named/ x8,r16,i8,c
      end

      subroutine s()
         complex(4) x8
         real(16) r16(2)
         integer(8) i8(2)
         character*5 c(2)
         common /named/ x8,r16,i8,c
!          x8    = (1.123456e0,2.123456e0)      ! promotion did not occur
!         r16(1) = 1.123q0                      ! no padding
!         r16(2) = 2.123q0                      ! no padding
!          i8(1) = 1000                         ! no padding
!          i8(2) = 2000                         ! no padding
!           c(1) = "abcde"                      ! no padding
!           c(2) = "12345"                      ! no padding
      end subroutine s
```

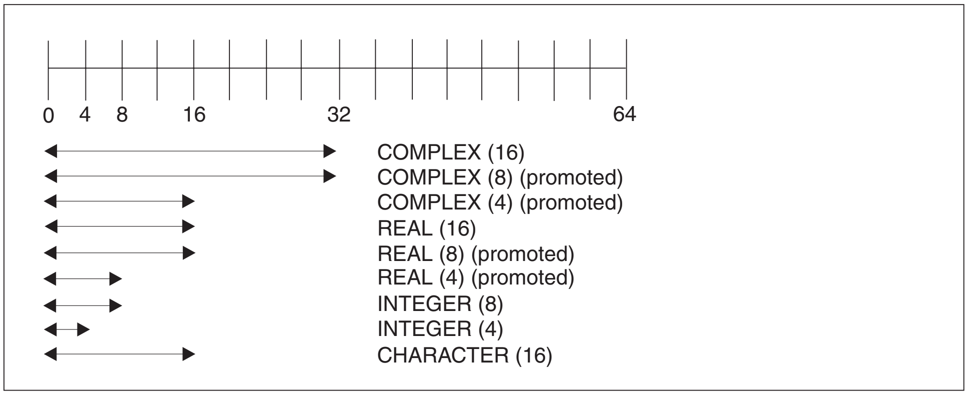


Figure 7. Storage Relationships with -qautodbl=dbl

```
@process autodbl(dbl)
      block data
      complex(4) x8
      real(16) r16(2)    /1.123q0,2.123q0/
      real(8) r8
      real(4) r4         /1.123456789e0/
      integer(8) i8(2)   /1000,2000/
      character*5 c(2)  /"abcde","12345"/
      equivalence (x8,r8)
      common /named/ r16,i8,c,r4
!     Storage relationship between r8 and x8 is preserved.
!     Data values are NOT preserved between r8 and x8.
      end

      subroutine s()
         real(16) r16(2)
         real(8) r4
         integer(8) i8(2)
         character*5 c(2)
         common /named/ r16,i8,c,r4
!         r16(1) = 1.123q0                        ! no padding
!         r16(2) = 2.123q0                        ! no padding
!          r4    = 1.123456789d0                  ! promotion occurred
!          i8(1) = 1000                           ! no padding
!          i8(2) = 2000                           ! no padding
!           c(1) = "abcde"                        ! no padding
!           c(2) = "12345"                        ! no padding
      end subroutine s
```

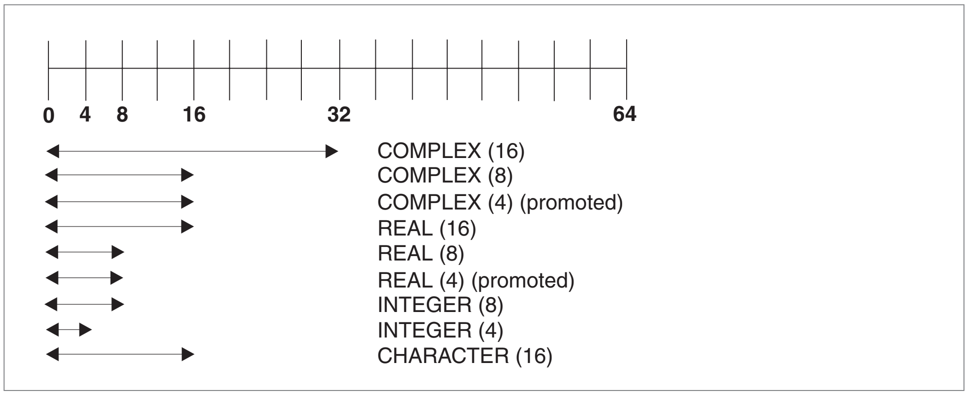


Figure 8. Storage Relationships with -qautobl=dbl4

```
@process autodbl(dbl4)
      complex(8) x16    /(1.123456789d0,2.123456789d0)/
      complex(4) x8
      real(4) r4(2)
      equivalence (x16,x8,r4)
!     Storage relationship between r4 and x8 is preserved.
!     Data values between r4 and x8 are preserved.
!     x16   = (1.123456789d0,2.123456789d0)      ! promotion did not occur
!     x8    = (1.123456789d0,2.123456789d0)      ! promotion occurred
!     r4(1) = 1.123456789d0                      ! promotion occurred
!     r4(2) = 2.123456789d0                      ! promotion occurred
      end
```

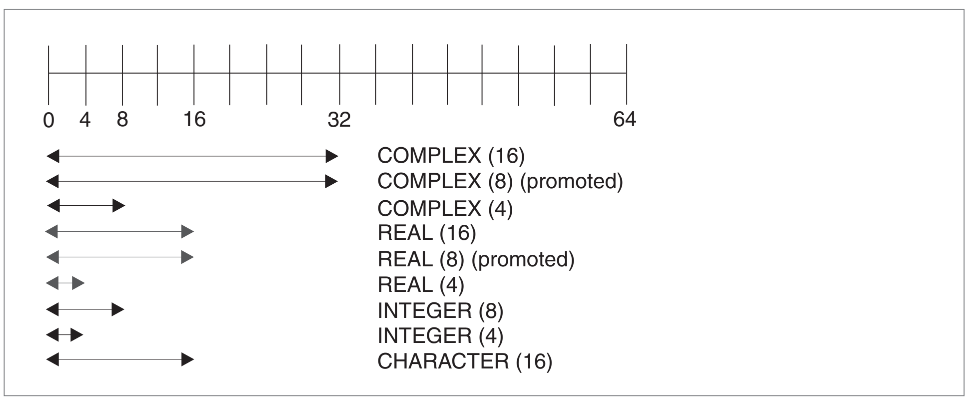


Figure 9. Storage Relationships with -qautodbl=dbl8

```
@process autodbl(dbl8)
      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)
!                                                ! promotion occurred
!     x8    = upper 8 bytes of r8(1)             ! promotion did not occur
!     r8(1) = 1.123456789123456789q0             ! promotion occurred
!     r8(2) = 2.123456789123456789q0             ! promotion occurred
      end
```

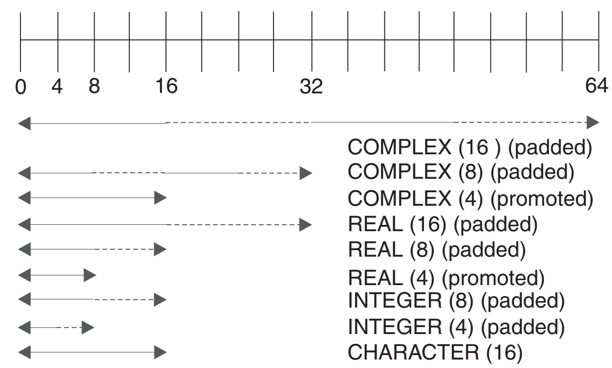


Figure 10. Storage Relationships with -qautodbl=dblpad4

In the figure above, the dashed lines represent the padding.

```
@process autodbl(dblpad4)
      complex(8) x16  /(1.123456789d0,2.123456789d0)/
      complex(4) x8
      real(4) r4(2)
      integer(8) i8(2)
      equivalence(x16,x8,r4,i8)
!     Storage relationship among all entities is preserved.
!     Date values between x8 and r4 are preserved.
!     x16   = (1.123456789d0,2.123456789d0)      ! padding occurred
!     x8    = (upper 8 bytes of x16, 8 byte pad) ! promotion occurred
!     r4(1) = real(x8)                           ! promotion occurred
!     r4(2) = imag(x8)                           ! promotion occurred
!     i8(1) = real(x16)                          ! padding occurred
!     i8(2) = imag(x16)                          ! padding occurred
      end
```

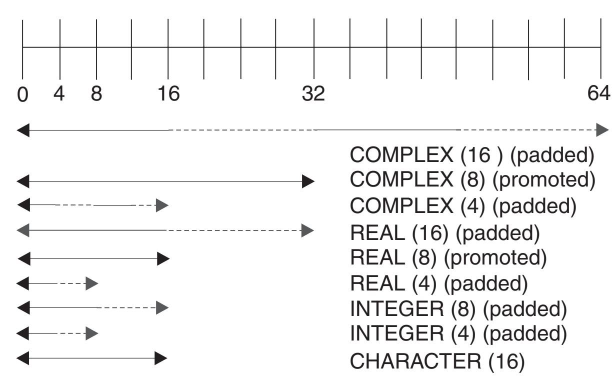


Figure 11. Storage Relationships with -qautodbl=dblpad8

In the figure above, the dashed lines represent the padding.

```
@process autodbl(dblpad8)
      complex(8) x16    /(1.123456789123456789d0,2.123456789123456789d0)/
      complex(4) x8
      real(8) r8(2)
      integer(8) i8(2)
      byte b(16)
      equivalence (x16,x8,r8,i8,b)
!     Storage relationship among all entities is preserved.
```

```
!     Data values between r8 and x16 are preserved.
!     Data values between i8 and b are preserved.
!     x16 = (1.123456789123456789q0,2.123456789123456789q0)
!                                             ! promotion occurred
!     x8 = upper 8 bytes of r8(1)             ! padding occurred
!     r8(1) = real(x16)                       ! promotion occurred
!     r8(2) = imag(x16)                       ! promotion occurred
!     i8(1) = upper 8 bytes of real(x16)      ! padding occurred
!     i8(2) = upper 8 bytes of imag(x16)      ! padding occurred
!     b(1:8)= i8(1)                           ! padding occurred
!     b(9:16)= i8(2)                          ! padding occurred
      end
```

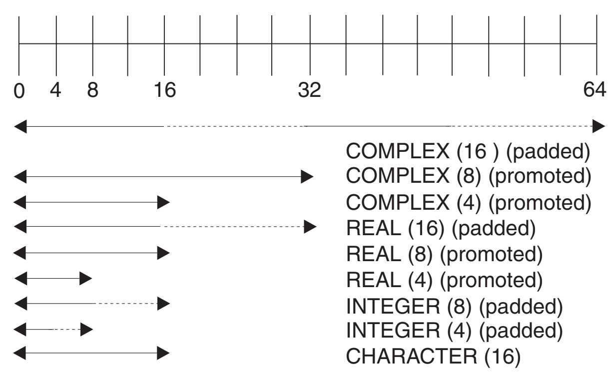


Figure 12. Storage Relationships with -qautodbl=dblpad

In the figure above, the dashed lines represent the padding.

```
@process autodbl(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
```

Appendix C. Using the Mathematical Acceleration Subsystem (MASS)

XL Fortran Enterprise Edition ships the Mathematical Acceleration Subsystem (MASS), a set of libraries of tuned mathematical intrinsic functions that provide improved performance over the corresponding standard math library functions. The accuracy and exception handling might not be identical in MASS functions and standard math library functions.

The MASS libraries on AIX consist of a library of scalar functions, described in "Using the Scalar Library," and a set of vector libraries tuned for specific architectures, described in "Using the Vector Libraries" on page 420. "Compiling and Linking a Program with MASS" on page 423 describes how to compile and link a program that uses the MASS libraries, and how to selectively use the MASS scalar library functions in concert with the regular intrinsic scalar functions.

Using the Scalar Library

The MASS scalar library, **libmass.a**, contains an accelerated set of frequently used math intrinsic functions in the system library. These functions all accept double-precision parameters and return a double-precision result, and are summarized in Table 33. To provide the interface declarations for the functions, include **math.include** in your source files.

Table 33. MASS Scalar Library Functions

Function	Description
sqrt	Returns the square root of x
rsqrt	Returns the reciprocal of the square root of x
exp	Returns the exponential function of x
log	Returns the natural logarithm of x
sin	Returns the sine of x
cos	Returns the cosine of x
tan	Returns the tangent of x
atan	Returns the arctangent of x
atan2	Returns the arctangent of x/y
sinh	Returns the hyperbolic sine of x
cosh	Returns the hyperbolic cosine of x
tanh	Returns the hyperbolic tangent of x
dnint	Returns the nearest integer to x (as a double)
x**y	Returns x raised to the power y

The following example shows interface declarations for some of the MASS scalar functions:

```
interface

real*8 function sqrt (x)
  real*8 x      ! Returns the square root of x.
```

```
end function sqrt

real*8 function rsqrt (x)
  real*8 x      ! Returns the reciprocal of the square root of x.
end function rsqrt

end interface
```

The trigonometric functions (**sin**, **cos**, **tan**) return NaN (Not-a-Number) values for large arguments (abs(x)>2**50*pi).

Note: In some cases, the MASS functions are not as accurate as those in the standard intrinsic functions and they may handle edge cases differently (**sqrt(Inf)**, for example).

Using the Vector Libraries

The MASS vector libraries are shipped in the following archives:

libmassv.a
The general vector library.

libmassvp3.a
Contains some functions that have been tuned for the POWER3 architecture. The remaining functions are identical to those in **libmassv.a**.

libmassvp4.a
Contains some functions that have been tuned for the POWER4 architecture. The remaining functions are identical to those in **libmassv.a**. If you are using POWER5, this library is the recommended choice.

With the exception of a few functions (described below), all of the functions in **libmassv.a**, **libmassvp3.a**, and **libmassvp4.a** accept three parameters:

- a double-precision or single-precision vector input parameter
- a double-precision or single-precision output parameter
- an integer vector-length parameter

These functions are all of the form:

function_name (*y*,*x*,*n*)

where *x* is the source vector, *y* is the target vector, and *n* is the vector length. The parameters *y* and *x* are assumed to be double-precision for functions whose prefix is **v**, and single-precision for functions with the prefix **vs**. As an example, the following code:

```
include 'massv.include'

real*8 x(500), y(500)
integer n
n = 500
...
call vexp (y, x, n)
```

outputs a vector *y* of length 500 whose elements are exp(x(i)), with i=1,...,500.

The single-precision and double-precision functions contained in the vector libraries are summarized in Table 34 on page 421. To provide the interface declarations for the functions, include **massv.include** in your source files.

Table 34. MASS Vector Library Functions

Double-precision function	Single-precision function	Arguments	Description
vacos	vsacos	`(y,x,n)`	Sets y(i) to the arccosine of x(i), for i=1,..,n
vasin	vsasin	`(y,x,n)`	Sets y(i) to the arcsine of x(i), for i=1,..,n
vatan2	vsatan2	`(z,x,y,n)`	Sets z(i) to the arctangent of x(i)/y(i), for i=1,..,n
vcos	vscos	`(y,x,n)`	Sets y(i) to the cosine of x(i), for i=1,..,n
vcosh	vscosh	`(y,x,n)`	Sets y(i) to the hyperbolic cosine of x(i), for i=1,..,n
vcosisin	vscosisin	`(y,x,n)`	Sets the real part of y(i) to the cosine of x(i) and the imaginary part of y(i) to the sine of x(i), for i=1,..,n
vdint		`(y,x,n)`	Sets y(i) to the integer truncation of x(i), for i=1,..,n
vdiv	vsdiv	`(z,x,y,n)`	Sets z(i) to x(i)/y(i), for i=1,..,n
vdnint		`(y,x,n)`	Sets y(i) to the nearest integer to x(i), for i=1,..,n
vexp	vsexp	`(y,x,n)`	Sets y(i) to the exponential function of x(i), for i=1,..,n
vexpm1	vsexpm1	`(y,x,n)`	Sets y(i) to (the exponential function of x(i))-1, for i=1,..,n
vlog	vslog	`(y,x,n)`	Sets y(i) to the natural logarithm of x(i), for i=1,..,n
vlog10	vslog10	`(y,x,n)`	Sets y(i) to the base-10 logarithm of x(i), for i=1,..,n
vlog1p	vslog1p	`(y,x,n)`	Sets y(i) to the natural logarithm of (x(i)+1), for i=1,..,n
vpow	vspow	`(z,x,y,n)`	Sets z(i) to x(i) raised to the power y(i), for i=1,..,n
vrec	vsrec	`(y,x,n)`	Sets y(i) to the reciprocal of x(i), for i=1,..,n
vrsqrt	vsrsqrt	`(y,x,n)`	Sets y(i) to the reciprocal of the square root of x(i), for i=1,..,n
vsin	vssin	`(y,x,n)`	Sets y(i) to the sine of x(i), for i=1,..,n
vsincos	vssincos	`(y,z,x,n)`	Sets y(i) to the sine of x(i) and z(i) to the cosine of x(i), for i=1,..,n
vsinh	vssinh	`(y,x,n)`	Sets y(i) to the hyperbolic sine of x(i), for i=1,..,n
vsqrt	vssqrt	`(y,x,n)`	Sets y(i) to the square root of x(i), for i=1,..,n
vtan	vstan	`(y,x,n)`	Sets y(i) to the tangent of x(i), for i=1,..,n

Table 34. MASS Vector Library Functions (continued)

vtanh	vstanh	(y,x,n)	Sets y(i) to the hyperbolic tangent of x(i), for i=1,..,n

The following example shows interface declarations for some of the MASS double-precision vector functions:

```
interface

subroutine vsqrt (y, x, n)
  real*8 y(*), x(*)
  integer n        ! Sets y(i) to the square root of x(i), for i=1,..,n
end subroutine vsqrt

subroutine vrsqrt (y, x, n)
  real*8 y(*), x(*)
  integer n        ! Sets y(i) to the reciprocal of the square root of x(i),
                   ! for i=1,..,n
end subroutine vrsqrt

end interface
```

The following example shows interface declarations for some of the MASS single-precision vector functions:

```
interface

subroutine vssqrt (y, x, n)
  real*4 y(*), x(*)
  integer n       ! Sets y(i) to the square root of x(i), for i=1,..,n
end subroutine vssqrt

subroutine vsrsqrt (y, x, n)
  real*4 y(*), x(*)
  integer n       ! Sets y(i) to the reciprocal of the square root of x(i),
                   ! for i=1,..,n
end subroutine vsrsqrt

end interface
```

The functions **vatan2**, **vdiv**, and **vpow** take four parameters and are of the form *function_name*(*z*,*x*,*y*,*n*). The function **vsincos** takes four parameters of the form *function_name*(*y*,*z*,*x*,*n*). The function **vatan2** outputs a vector *z* whose elements are atan(x(i)/y(i)). The function **vdiv** outputs a vector *z* whose elements are x(i)/y(i), with i=1,..,n. The function **vsincos** outputs two vectors, *y* and *z*, whose elements are sin(x(i)) and cos(x(i)) respectively.

In **vcosisin(y,x,n)**, *x* is a vector of *n* **real*8** elements and the function outputs a vector *y* of *n* **complex*16** elements of the form (cos(x(i)),sin(x(i))).

Consistency of MASS Vector Functions

In the interest of speed, the MASS libraries make certain trade-offs. One of these involves the consistency of certain MASS vector functions. For certain functions, it is possible that the result computed for a particular input value will vary slightly (usually only in the least significant bit) depending on its position in the vector, the vector length, and nearby elements of the input vector. Also, the results produced by the different MASS libraries are not necessarily bit-wise identical.

However, the **libmassvp4.a** library provides newer, consistent versions of certain functions. These consistent functions are: **vsqrt**, **vssqrt**, **vlog**, **vrec**, **vdiv**, **vexp**, **vsin**, **vcos**, **vacos**, **vasin**, **vatan2**, **vrsqrt**, **vscos**, **vsdiv**, **vsrec**, **vssin**.

The accuracy of the vector functions is comparable to that of the corresponding scalar functions in libmass.a, though results may not be bit-wise identical.

For more information on consistency and avoiding inconsistency with the vector libraries, as well as performance and accuracy data, see the MASS Web site at URL `http://www.ibm.com/software/awdtools/fortran/xlfortran/mass`.

Compiling and Linking a Program with MASS

To compile an application that calls the routines in these libraries, specify **mass** and **massv** (or **massvp3** or **massvp4**) on the **-l** linker option. For example, if the MASS libraries are installed in the default directory, you could specify:

```
xlf progf.f -o progf -lmass -lmassv
```

The MASS functions must run in the round-to-nearest rounding mode and with floating-point exception trapping disabled. (These are the default compilation settings.)

Using libmass.a with the Standard Intrinsic Functions

If you wish to use the **libmass.a** scalar library for some functions and the normal intrinsic functions, follow this procedure to compile and link your program:

1. Create an export list (this can be a flat text file) containing the names of the desired functions. For example, to select only the fast tangent function from **libmass.a** for use with the program `sample.f`, create a file called `fast_tan.exp` with the following line:
   ```
   tan
   ```
2. Create a shared object from the export list with the **ld** command, linking with the **libmass.a** library. For example:
   ```
   ld -bexport:fast_tan.exp -o fast_tan.o -bnoentry -lmass -bmodtype:SRE
   ```
3. Archive the shared object into a library with the **ar** command. For example:
   ```
   ar -q libfasttan.a fast_tan.o
   ```
4. Create the final executable using **xlf**. This links only the functions specified in the object file (in this example, the **tan** function) and the remainder of the math functions from the standard system library. For example:
   ```
   xlf -o sample sample.f -Ldir_containing_libfasttan.a -lfasttan
   ```

Note: The MASS **cos** function is automatically linked if you export MASS **sin**; MASS **atan2** is automatically linked if you export MASS **atan**.

Related Information

- **ld** and **ar** in *AIX Commands Reference*

Appendix D. XL Fortran Internal Limits

Language Feature	Limit
Maximum number of iterations performed by **DO** loops with loop control with index variable of type INTEGER(n) for *n* = 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	6 700
Maximum length of a free source form statement	6 700
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** run-time option for the record length to be greater than (2**31)-1 and up to (2**63)-1. If you use the default **uwidth=32** run-time option, the maximum length of a record in an unformatted sequential file is (2**31)-1 bytes.
Allowable bound range of an array dimension	The bound of an array dimension can be positive, negative, or zero within the range -(2**31) to 2**31-1 in 32-bit mode, or -(2**63) to 2**63-1 in 64-bit mode.
Allowable external unit numbers	0 to (2**31)-1 **2**
Maximum numeric format field width	2 000
Maximum number of concurrent open files	2 000 **3**

1 You can have as many continuation lines as you need to create a statement with a maximum of 6 700 bytes.

2 The value must be representable in an **INTEGER(4)** object, even if specified by an **INTEGER(8)** variable.
3 In practice, this value is somewhat lower because of files that the run-time system may open, such as the preconnected units 0, 5, and 6.

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504–1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2–31 Roppongi 3–chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION ″AS IS″ WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Lab Director
IBM Canada Limited
8200 Warden Avenue
Markham, Ontario, Canada
L6G 1C7

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

This software and documentation are based in part on the Fourth Berkeley Software Distribution under license from the Regents of the University of California. We acknowledge the following institution for its role in this product's development: the Electrical Engineering and Computer Sciences Department at the Berkeley campus.

OpenMP is a trademark of the OpenMP Architecture Review Board. Portions of this document may have been derived from the *OpenMP Fortran Language Application Program Interface, Version 2.0* (November 2000) specification. Copyright 1997-2000 OpenMP Architecture Review Board.

Programming Interface Information

Programming interface information is intended to help you create application software using this program.

General-use programming interfaces allow the customer to write application software that obtain the services of this program's tools.

However, this information may also contain diagnosis, modification, and tuning information. Diagnosis, modification, and tuning information is provided to help you debug your application software.

Note: Do not use this diagnosis, modification, and tuning information as a programming interface because it is subject to change.

Trademarks and Service Marks

The following terms, used in this publication, are trademarks or service marks of the International Business Machines Corporation in the United States or other countries or both:

AIX	IBM	PowerPC
PowerPC 601	PowerPC 603	PowerPC 604
POWER	POWER2	POWER3
POWER4	POWER5	POWERserver
POWER2 Architecture	RISC System/6000	RS/6000
SAA	z/OS	

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

Windows is a trademark of Microsoft Corporation in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.

Glossary

This glossary defines terms that are commonly used in this document. It includes definitions developed by the American National Standards Institute (ANSI) and entries from the *IBM Dictionary of Computing*.

A

active processor. See *online processor*.

actual argument. An expression, variable, procedure, or alternate return specifier that is specified in a procedure reference.

alias. A single piece of storage that can be accessed through more than a single name. Each name is an alias for that storage.

alphabetic character. A letter or other symbol, excluding digits, used in a language. Usually the uppercase and lowercase letters A through Z plus other special symbols (such as $ and _) allowed by a particular language.

alphanumeric. Pertaining to a character set that contains letters, digits, and usually other characters, such as punctuation marks and mathematical symbols.

American National Standard Code for Information Interchange. See *ASCII*.

argument. An expression that is passed to a function or subroutine. See also *actual argument*, *dummy argument*.

argument association. The relationship between an actual argument and a dummy argument during the invoking of a procedure.

arithmetic constant. A constant of type integer, real, or complex.

arithmetic expression. One or more arithmetic operators and arithmetic primaries, the evaluation of which produces a numeric value. An arithmetic expression can be an unsigned arithmetic constant, the name of an arithmetic constant, or a reference to an arithmetic variable, function reference, or a combination of such primaries formed by using arithmetic operators and parentheses.

arithmetic operator. A symbol that directs the performance of an arithmetic operation. The intrinsic arithmetic operators are:

```
+      addition
-      subtraction
*      multiplication
/      division
**     exponentiation
```

array. An entity that contains an ordered group of scalar data. All objects in an array have the same data type and type parameters.

array declarator. The part of a statement that describes an array used in a program unit. It indicates the name of the array, the number of dimensions it contains, and the size of each dimension.

array element. A single data item in an array, identified by the array name and one or more subscripts. See also *subscript*.

array name. The name of an ordered set of data items.

array section. A subobject that is an array and is not a structure component.

ASCII. The standard code, using a coded character set consisting of 7-bit coded characters (8-bits including parity check), that is used for information interchange among data processing systems, data communication systems, and associated equipment. The ASCII set consists of control characters and graphic characters. See also *Unicode*.

asynchronous. Pertaining to events that are not synchronized in time or do not occur in regular or predictable time intervals. For example, input events are controlled by the user; the program can read them later.

assignment statement. An executable statement that defines or redefines a variable based on the result of expression evaluation.

associate name. The name by which a selector of an **ASSOCIATE** construct is known within the construct.

attribute. A property of a data object that may be specified in a type declaration statement, attribute specification statement, or through a default setting.

automatic parallelization. The process by which the compiler attempts to parallelize both explicitly coded **DO** loops and **DO** loops generated by the compiler for array language.

B

binary constant. A constant that is made of one or more binary digits (0 and 1).

bind. To relate an identifier to another object in a program; for example, to relate an identifier to a value, an address or another identifier, or to associate formal parameters and actual parameters.

blank common. An unnamed common block.

block data subprogram. A subprogram headed by a **BLOCK DATA** statement and used to initialize variables in named common blocks.

bss storage. Uninitialized static storage.

busy-wait. The state in which a thread keeps executing in a tight loop looking for more work once it has completed all of its work and there is no new work to do.

byte constant. A named constant that is of type byte.

byte type. A data type representing a one-byte storage area that can be used wherever a **LOGICAL(1)**, **CHARACTER(1)**, or **INTEGER(1)** can be used.

C

character constant. A string of one or more alphabetic characters enclosed in apostrophes or double quotation marks.

character expression. A character object, a character-valued function reference, or a sequence of them separated by the concatenation operator, with optional parentheses.

character operator. A symbol that represents an operation, such as concatenation (//), to be performed on character data.

character set. All the valid characters for a programming language or for a computer system.

character string. A sequence of consecutive characters.

character substring. A contiguous portion of a character string.

character type. A data type that consists of alphanumeric characters. See also *data type*.

chunk. A subset of consecutive loop iterations.

collating sequence. The sequence in which the characters are ordered for the purpose of sorting, merging, comparing, and processing indexed data sequentially.

comment. A language construct for the inclusion of text in a program that has no effect on the execution of the program.

common block. A storage area that may be referred to by a calling program and one or more subprograms.

compile. To translate a source program into an executable program (an object program).

compiler directive. Source code that controls what XL Fortran does rather than what the user program does.

complex constant. An ordered pair of real or integer constants separated by a comma and enclosed in parentheses. The first constant of the pair is the real part of the complex number; the second is the imaginary part.

complex number. A number consisting of an ordered pair of real numbers, expressible in the form **a+bi**, where **a** and **b** are real numbers and **i** squared equals -1.

complex type. A data type that represents the values of complex numbers. The value is expressed as an ordered pair of real data items separated by a comma and enclosed in parentheses. The first item represents the real part of the complex number; the second represents the imaginary part.

conform. To adhere to a prevailing standard. An executable program conforms to the Fortran 95 Standard if it uses only those forms and relationships described therein and if the executable program has an interpretation according to the Fortran 95 Standard. A program unit conforms to the Fortran 95 Standard if it can be included in an executable program in a manner that allows the executable program to be standard-conforming. A processor conforms to the standard if it executes standard-conforming programs in a manner that fulfills the interpretations prescribed in the standard.

connected unit. In XL Fortran, a unit that is connected to a file in one of three ways: explicitly via the **OPEN** statement to a named file, implicitly, or by preconnection.

constant. A data object with a value that does not change. The four classes of constants specify numbers (arithmetic), truth values (logical), character data (character), and typeless data (hexadecimal, octal, and binary). See also *variable*.

construct. A sequence of statements starting with a **SELECT CASE**, **DO**, **IF**, or **WHERE** statement and ending with the corresponding terminal statement.

continuation line. A line that continues a statement beyond its initial line.

control statement. A statement that is used to alter the continuous sequential invocation of statements; a control statement may be a conditional statement, such as **IF**, or an imperative statement, such as **STOP**.

D

data object. A variable, constant, or subobject of a constant.

data striping. Spreading data across multiple storage devices so that I/O operations can be performed in parallel for better performance. Also known as *disk striping*.

data transfer statement. A **READ**, **WRITE**, or **PRINT** statement.

data type. The properties and internal representation that characterize data and functions. The intrinsic types are integer, real, complex, logical, and character. See also *intrinsic*.

debug line. Allowed only for fixed source form, a line containing source code that is to be used for debugging. Debug lines are defined by a `D` or `X` in column 1. The handling of debug lines is controlled by the **-qdlines** and **-qxlines** compiler options.

default initialization. The initialization of an object with a value specified as part of a derived type definition.

definable variable. A variable whose value can be changed by the appearance of its name or designator on the left of an assignment statement.

delimiters. A pair of parentheses or slashes (or both) used to enclose syntactic lists.

denormalized number. An IEEE number with a very small absolute value and lowered precision. A denormalized number is represented by a zero exponent and a non-zero fraction.

derived type. A type whose data have components, each of which is either of intrinsic type or of another derived type.

digit. A character that represents a nonnegative integer. For example, any of the numerals from 0 through 9.

directive. A type of comment that provides instructions and information to the compiler.

disk striping. See *data striping*.

DO loop. A range of statements invoked repetitively by a **DO** statement.

DO variable. A variable, specified in a **DO** statement, that is initialized or incremented prior to each occurrence of the statement or statements within a **DO** range. It is used to control the number of times the statements within the range are executed.

DOUBLE PRECISION constant. A constant of type real with twice the precision of the default real precision.

dummy argument. An entity whose name appears in the parenthesized list following the procedure name in a **FUNCTION**, **SUBROUTINE**, **ENTRY**, or statement function statement.

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

dynamic extent. For a directive, the lexical extent of the directive and all subprograms called from within the lexical extent.

E

edit descriptor. An abreviated keyword that controls the formatting of integer, real, or complex data.

elemental. Pertaining to an intrinsic operation, procedure or assignment that is applied independently to elements of an array or corresponding elements of a set of conformable arrays and scalars.

embedded blank. A blank that is surrounded by any other characters.

entity. A general term for any of the following: a program unit, procedure, operator, interface block, common block, external unit, statement function, type, named variable, expression, component of a structure, named constant, statement label, construct, or namelist group.

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

executable program. A program that can be executed as a self-contained procedure. It consists of a main program and, optionally, modules, subprograms and non-Fortran external procedures.

executable statement. A statement that causes an action to be taken by the program; for example, to perform a calculation, test conditions, or alter normal sequential execution.

explicit initialization. The initialization of an object with a value stated in a data statement initial value list, block data program unit, type declaration statement, or array constructor.

explicit interface. For a procedure referenced in a scoping unit, the property of being an internal procedure, module procedure, intrinsic procedure, external procedure that has an interface block, recursive procedure reference in its own scoping unit, or dummy procedure that has an interface block.

expression. A sequence of operands, operators, and parentheses. It may be a variable, a constant, or a function reference, or it may represent a computation.

extended-precision constant. A processor approximation to the value of a real number that occupies 16 consecutive bytes of storage.

external file. A sequence of records on an input/output device. See also *internal file*.

external name. The name of a common block, subroutine, or other global procedure, which the linker uses to resolve references from one compilation unit to another.

external procedure. A procedure that is defined by an external subprogram or by a means other than Fortran.

F

field. An area in a record used to contain a particular category of data.

file. A sequence of records. See also *external file*, *internal file*.

file index. See *i-node*.

floating-point number. A real number represented by a pair of distinct numerals. The real number is the product of the fractional part, one of the numerals, and a value obtained by raising the implicit floating-point base to a power indicated by the second numeral.

format. (1) A defined arrangement of such things as characters, fields, and lines, usually used for displays, printouts, or files. (2) To arrange such things as characters, fields, and lines.

formatted data. Data that is transferred between main storage and an input/output device according to a specified format. See also *list-directed* and *unformatted record*.

function. A procedure that returns the value of a single variable or an object and that usually has a single exit. See also *intrinsic procedure*, *subprogram*.

G

generic identifier. A lexical token that appears in an **INTERFACE** statement and is associated with all the procedures in an interface block.

H

hard limit. A system resource limit that can only be raised or lowered by using root authority, or cannot be altered because it is inherent in the system or operating environments's implementation. See also *soft limit*.

hexadecimal. Pertaining to a system of numbers to the base sixteen; hexadecimal digits range from 0 (zero) through 9 (nine) and A (ten) through F (fifteen).

hexadecimal constant. A constant, usually starting with special characters, that contains only hexadecimal digits.

high order transformations. A type of optimization that restructures loops.

Hollerith constant. A string of any characters capable of representation by XL Fortran and preceded with *n***H**, where *n* is the number of characters in the string.

host. A main program or subprogram that contains an internal procedure is called the host of the internal procedure. A module that contains a module procedure is called the host of the module procedure.

host association. The process by which an internal subprogram, module subprogram, or derived-type definition accesses the entities of its host.

I

IPA. Interprocedural analysis, a type of optimization that allows optimizations to be performed across procedure boundaries and across calls to procedures in separate source files.

implicit interface. A procedure referenced in a scoping unit other than its own is said to have an implicit interface if the procedure is an external procedure that does not have an interface block, a dummy procedure that does not have an interface block, or a statement function.

implied DO. An indexing specification (similar to a **DO** statement, but without specifying the word **DO**) with a list of data elements, rather than a set of statements, as its range.

infinity. An IEEE number (positive or negative) created by overflow or division by zero. Infinity is represented by an exponent where all the bits are 1's, and a zero fraction.

i-node. The internal structure that describes the individual files in the operating system. There is one i-node for each file. An i-node contains the node, type, owner, and location of a file. A table of i-nodes is stored near the beginning of a file system. Also known as *file index*.

input/output (I/O). Pertaining to either input or output, or both.

input/output list. A list of variables in an input or output statement specifying the data to be read or written. An output list can also contain a constant, an expression involving operators or function references, or an expression enclosed in parentheses.

integer constant. An optionally signed digit string that contains no decimal point.

interface block. A sequence of statements from an **INTERFACE** statement to the corresponding **END INTERFACE** statement.

interface body. A sequence of statements in an interface block from a **FUNCTION** or **SUBROUTINE** statement to the corresponding **END** statement.

interference. A situation in which two iterations within a **DO** loop have dependencies upon one another.

internal file. A sequence of records in internal storage. See also *external file*.

interprocedural analysis. See *IPA*.

intrinsic. Pertaining to types, operations, assignment statements, and procedures that are defined by Fortran language standards and can be used in any scoping unit without further definition or specification.

intrinsic module. A module that is provided by the compiler and is available to any program.

intrinsic procedure. A procedure that is provided by the compiler and is available to any program.

K

keyword. (1) A statement keyword is a word that is part of the syntax of a statement (or directive) and that may be used to identify the statement. (2) An argument keyword specifies a name for a dummy argument.

kind type parameter. A parameter whose values label the available kinds of an intrinsic type.

L

lexical extent. All of the code that appears directly within a directive construct.

lexical token. A sequence of characters with an indivisible interpretation.

link-edit. To create a loadable computer program by means of a linker.

linker. A program that resolves cross-references between separately compiled or assembled object modules and then assigns final addresses to create a single relocatable load module. If a single object module is linked, the linker simply makes it relocatable.

list-directed. A predefined input/output format that depends on the type, type parameters, and values of the entities in the data list.

literal. A symbol or a quantity in a source program that is itself data, rather than a reference to data.

literal constant. A lexical token that directly represents a scalar value of intrinsic type.

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

logical constant. A constant with a value of either `true` or `false` (or T or F).

logical operator. A symbol that represents an operation on logical expressions:

```
.NOT.    (logical negation)
.AND.    (logical conjunction)
.OR.     (logical union)
.EQV.    (logical equivalence)
.NEQV.   (logical nonequivalence)
.XOR.    (logical exclusive disjunction)
```

loop. A statement block that executes repeatedly.

License Use Management (LUM). A run-time license-management application based on Gradient Technologies' Version 2.0.1 (Version 1.1.2a) of the Network Licensing System. The system allows software vendors to bundle compliance mechanisms with their software. In tracking license usage, License Use Management allows customers to easily comply with their software license agreements.

M

_main. The default name given to a main program by the compiler if the main program was not named by the programmer.

main program. The first program unit to receive control when a program is run. See also *subprogram*.

master thread. The head process of a group of threads.

module. A program unit that contains or accesses definitions to be accessed by other program units.

mutex. A primitive object that provides mutual exclusion between threads. A mutex is used cooperatively between threads to ensure that only one of the cooperating threads is allowed to access shared data or run certain application code at a time.

N

NaN (not-a-number). A symbolic entity encoded in floating-point format that does not correspond to a number. See also *quiet NaN*, *signalling NaN*.

name. A lexical token consisting of a letter followed by up to 249 alphanumeric characters (letters, digits, and underscores). Note that in FORTRAN 77, this was called a symbolic name.

named common. A separate, named common block consisting of variables.

namelist group name. The first parameter in the NAMELIST statement that names a list of names to be used in READ, WRITE, and PRINT statements.

negative zero. An IEEE representation where the exponent and fraction are both zero, but the sign bit is 1. Negative zero is treated as equal to positive zero.

nest. To incorporate a structure or structures of some kind into a structure of the same kind. For example, to nest one loop (the nested loop) within another loop (the nesting loop); to nest one subroutine (the nested subroutine) within another subroutine (the nesting subroutine).

nonexecutable statement. A statement that describes the characteristics of a program unit, data, editing information, or statement functions, but does not cause any action to be taken by the program.

nonexisting file. A file that does not physically exist on any accessible storage medium.

normal. A floating point number that is not denormal, infinity, or NaN.

not-a-number. See *NaN*.

numeric constant. A constant that expresses an integer, real, complex, or byte number.

O

octal. Pertaining to a system of numbers to the base eight; the octal digits range from 0 (zero) through 7 (seven).

octal constant. A constant that is made of octal digits.

one-trip DO-loop. A **DO** loop that is executed at least once, if reached, even if the iteration count is equal to 0. (This type of loop is from FORTRAN 66.)

online processor. In a multiprocessor machine, a processor that has been activated (brought online). The number of online processors is less than or equal to the number of physical processors actually installed in the machine. Also known as *active processor*.

operator. A specification of a particular computation involving one or two operands.

P

pad. To fill unused positions in a field or character string with dummy data, usually zeros or blanks.

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

PDF. See *profile-directed feedback*.

pointee array. An explicit-shape or assumed-size array that is declared in an integer **POINTER** statement or other specification statement.

pointer. A variable that has the **POINTER** attribute. A pointer must not be referenced or defined unless it is pointer associated with a target. If it is an array, it does not have a shape unless it is pointer-associated.

preconnected file. A file that is connected to a unit at the beginning of execution of the executable program. Standard error, standard input, and standard output are preconnected files (units 0, 5 and 6, respectively).

predefined convention. The implied type and length specification of a data object, based on the initial character of its name when no explicit specification is given. The initial characters I through N imply type integer of length 4; the initial characters A through H, O through Z, $, and _ imply type real of length 4.

present. A dummy argument is present in an instance of a subprogram if it is associated with an actual argument and the actual argument is a dummy argument that is present in the invoking procedure or is not a dummy argument of the invoking procedure.

primary. The simplest form of an expression: an object, array constructor, structure constructor, function reference, or expression enclosed in parentheses.

procedure. A computation that may be invoked during program execution. It may be a function or a subroutine. It may be an intrinsic procedure, an external procedure, a module procedure, an internal procedure, a dummy procedure, or a statement function. A subprogram may define more than one procedure if it contains **ENTRY** statements.

profile-directed feedback (PDF). A type of optimization that uses information collected during application execution to improve performance of conditional branches and in frequently executed sections of code.

program unit. A main program or subprogram.

pure. An attribute of a procedure that indicates there are no side effects.

Q

quiet NaN. A NaN (not-a-number) value that does not signal an exception. The intent of a quiet NaN is to propagate a NaN result through subsequent computations. See also *NaN*, *signalling NaN*.

R

random access. An access method in which records can be read from, written to, or removed from a file in any order. See also *sequential access*.

rank. The number of dimensions of an array.

real constant. A string of decimal digits that expresses a real number. A real constant must contain a decimal point, a decimal exponent, or both.

record. A sequence of values that is treated as a whole within a file.

relational expression. An expression that consists of an arithmetic or character expression, followed by a relational operator, followed by another arithmetic or character expression.

relational operator. The words or symbols used to express a relational condition or a relational expression:

```
.GT.       greater than
.GE.       greater than or equal to
.LT.       less than
.LE.       less than or equal to
.EQ.       equal to
.NE.       not equal to
```

result variable. The variable that returns the value of a function.

return specifier. An argument specified for a statement, such as CALL, that indicates to which statement label control should return, depending on the action specified by the subroutine in the RETURN statement.

S

scalar. (1) A single datum that is not an array. (2) Not having the property of being an array.

scale factor. A number indicating the location of the decimal point in a real number (and, on input, if there is no exponent, the magnitude of the number).

scope. That part of an executable program within which a lexical token has a single interpretation.

scope attribute. That part of an executable program within which a lexical token has a single interpretation of a particular named property or entity.

scoping unit. (1) A derived-type definition. (2) An interface body, excluding any derived-type definitions and interface bodies contained within it. (3) A program unit or subprogram, excluding derived-type definitions, interface bodies, and subprograms contained within it.

selector. The object that is associated with the associate name in an **ASSOCIATE** construct.

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

sequential access. An access method in which records are read from, written to, or removed from a file based on the logical order of the records in the file. See also *random access*.

signalling NaN. A NaN (not-a-number) value that signals an invalid operation exception whenever it appears as an operand. The intent of the signaling NaN is to catch program errors, such as using an uninitialized variable. See also *NaN*, *quiet NaN*.

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

SMP. See *symmetric multiprocessing*.

soft limit. A system resource limit that is currently in effect for a process. The value of a soft limit can be raised or lowered by a process, without requiring root authority. The soft limit for a resource cannot be raised above the setting of the hard limit. See also *hard limit*.

spill space. The stack space reserved in each subprogram in case there are too many variables to hold in registers and the program needs temporary storage for register contents.

specification statement. A statement that provides information about the data used in the source program. The statement could also supply information to allocate data storage.

stanza. A group of lines in a file that together have a common function or define a part of the system. Stanzas are usually separated by blank lines or colons, and each stanza has a name.

statement. A language construct that represents a step in a sequence of actions or a set of declarations. Statements fall into two broad classes: executable and nonexecutable.

statement function. A name, followed by a list of dummy arguments, that is equated with an intrinsic or derived-type expression, and that can be used as a substitute for the expression throughout the program.

statement label. A number from one through five digits that is used to identify a statement. Statement labels can be used to transfer control, to define the range of a **DO**, or to refer to a **FORMAT** statement.

storage association. The relationship between two storage sequences if a storage unit of one is the same as a storage unit of the other.

structure. A scalar data object of derived type.

structure component. The part of a data object of derived-type corresponding to a component of its type.

subobject. A portion of a named data object that may be referenced or defined independently of other portions. It can be an array element, array section, structure component, or substring.

subprogram. A function subprogram or a subroutine subprogram. Note that in FORTRAN 77, a block data program unit was called a subprogram. See also *main program*.

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

subscript. A subscript quantity or set of subscript quantities enclosed in parentheses and used with an array name to identify a particular array element.

substring. A contiguous portion of a scalar character string. (Although an array section can specify a substring selector, the result is not a substring.)

symmetric multiprocessing (SMP). A system in which functionally-identical multiple processors are used in parallel, providing simple and efficient load-balancing.

synchronous. Pertaining to an operation that occurs regularly or predictably with regard to the occurrence of a specified event in another process.

syntax. The rules for the construction of a statement. See also *semantics*.

T

target. A named data object specified to have the **TARGET** attribute, a data object created by an **ALLOCATE** statement for a pointer, or a subobject of such an object.

thread. A stream of computer instructions that is in control of a process. A multithread process begins with one stream of instructions (one thread) and may later create other instruction streams to perform tasks.

thread visible variable. A variable that can be accessed by more than one thread.

time slice. An interval of time on the processing unit allocated for use in performing a task. After the interval has expired, processing unit time is allocated to another task, so a task cannot monopolize processing unit time beyond a fixed limit.

token. In a programming language, a character string, in a particular format, that has some defined significance.

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

type declaration statement. A statement that specifies the type, length, and attributes of an object or function. Objects can be assigned initial values.

U

unformatted record. A record that is transmitted unchanged between internal and external storage.

Unicode. A universal character encoding standard that supports the interchange, processing, and display of text that is written in any of the languages of the modern world. It also supports many classical and historical texts in a number of languages. The Unicode standard has a 16-bit international character set defined by ISO 10646. See also *ASCII*.

unit. A means of referring to a file to use in input/output statements. A unit can be connected or not connected to a file. If connected, it refers to a file. The connection is symmetric: that is, if a unit is connected to a file, the file is connected to the unit.

unsafe option. Any option that could result in grossly incorrect results if used in the incorrect context. Other options may result in very small variations from the default result, which is usually acceptable. Typically, using an unsafe option is an assertion that your code is not subject to the conditions that make the option unsafe.

use association. The association of names in different scoping units specified by a **USE** statement.

V

variable. A data object whose value can be defined and redefined during the execution of an executable program. It may be a named data object, array element, array section, structure component, or substring. Note that in FORTRAN 77, a variable was always scalar and named.

X

XPG4. X/Open Common Applications Environment (CAE) Portability Guide Issue 4; a document which defines the interfaces of the X/Open Common Applications Environment that is a superset of

POSIX.1-1990, POSIX.2-1992, and POSIX.2a-1992 containing extensions to POSIX standards from XPG3.

Z

zero-length character. A character object that has a length of 0 and is always defined.

zero-sized array. An array that has a lower bound that is greater than its corresponding upper bound. The array is always defined.

INDEX

Special characters

Numerics

A

B

C

G

H

I

J

K

O

P

Q

R

S

T

U

V

Program Number: 5724-I08

SC09-7898-00