Optimization and Programming Guide

Version 12.1
Note

Before using this information and the product it supports, read the information in "Notices" on page 289.

First edition

This edition applies to IBM XL Fortran for AIX, V12.1 (Program number 5724-U82) and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

© Copyright International Business Machines Corporation 1996, 2008. All rights reserved.

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

**About this information** ........................................ vii
- Who should read this information ................................ vii
- How to use this information ...................................... vii
- How this information is organized ................................. vii

**Conventions** ..................................................... viii

**Related information** ............................................. xii
- IBM XL Fortran information ....................................... xii
- Standards and specifications ...................................... xiii
- Other IBM information ........................................... xiv

**Technical support** ................................................ xiv
- How to send your comments ....................................... xiv

## Chapter 1. Optimizing your applications  .................................. 1

**Distinguishing between optimization and tuning** .......................... 1

**Steps in the optimization process** ..................................... 2

**Basic optimization** .................................................. 2
- Optimizing at level 0 ................................................ 3
- Optimizing at level 2 ................................................ 3

**Advanced optimization** ............................................. 4
- Optimizing at level 3 ................................................ 5
- An intermediate step: adding -qhot suboptions at level 3 .......... 6
- Optimizing at level 4 ................................................ 6
- Optimizing at level 5 ................................................ 7

**Specialized optimization techniques** .................................... 8
- High-order transformation (HOT) ................................... 8
- Interprocedural analysis (IPA) ...................................... 10
- Profile-directed feedback (PDF) .................................... 13

**Debugging optimized code** .......................................... 15
- Understanding different results in optimized programs .......... 16
- Debugging before optimization ................................... 16
- Using -qoptdebug to help debug optimized programs .......... 17

**Getting more performance** .......................................... 20

**Beyond performance: effective programming techniques** .............. 20

## Chapter 2. Tuning XL compiler applications  .................................. 21

**Tuning for your target architecture** .................................. 21
- Using -qarch ......................................................... 22
- Using -qtune ......................................................... 24
- Using -qcache ....................................................... 25

**Before you finish tuning** ............................................ 25

**Further option driven tuning** ....................................... 25
- Options for providing application characteristics ............... 26
- Options to control optimization transformations ................... 28
- Options to assist with performance analysis ....................... 29
- Options that can inhibit performance ............................. 30

## Chapter 3. Advanced optimization concepts  .................................. 33

**Aliasing** ............................................................. 33

**Inlining** ............................................................. 33
- Finding the right level of inlining .................................. 34

## Chapter 4. Managing code size ........................................... 37

**Steps for reducing code size** ....................................... 38

**Compiler option influences on code size** ............................. 38
- The -qipa compiler option ........................................ 38
- The -Q inlining option ............................................. 38
- The -qhot compiler option ......................................... 39
- The -qcompact compiler option .................................... 39

**Other influences on code size** ....................................... 39
- High activity areas .................................................. 39
- Computed GOTOs and CASE constructs ............................. 39
- Code size with dynamic or static linking .......................... 40

## Chapter 5. Compiler-friendly programming techniques ..................... 43

**General practices** ................................................... 43

**Variables and pointers** ............................................. 43

**Arrays** ............................................................ 44

**Choosing appropriate variable sizes** ................................ 44

## Chapter 6. High performance libraries  ...................................... 45

**Using the Mathematical Acceleration Subsystem libraries (MASS)** .... 45
- Using the scalar library ............................................. 45
- Using the vector libraries ......................................... 47

**Compiling and linking a program with MASS** .......................... 52
- Using the Basic Linear Algebra Subprograms – BLAS .............. 53
- BLAS function syntax .............................................. 53
- Linking the libxmlot library ....................................... 55

## Chapter 7. Parallel programming with XL Fortran  ............................... 57

**Compiling your SMP code** .......................................... 57
- The _OPENMP_C preprocessor macro and conditional compilation .... 57

**Setting OMP and SMP run time options** ................................ 58
- The XLSMPOPTS environment variable ............................ 58
- OpenMP environment variables .................................... 63

**Optimizing your SMP code** ......................................... 65
- Developing and running SMP applications .......................... 65

**An introduction to SMP directives** ................................ 66
- Parallel region construct .......................................... 66
- Work-sharing constructs ............................................ 66
- Combined parallel work-sharing constructs ....................... 66
- Synchronization constructs ........................................ 66
- Other OpenMP Directives ......................................... 67
- Non-OpenMP SMP directives ....................................... 67

**Detailed descriptions of SMP directives** .............................. 67
- ATOMIC ............................................................ 67
- BARRIER .......................................................... 70
- CRITICAL / END CRITICAL ....................................... 71
DO / END DO .................................. 73
DO SERIAL ................................ 76
FLUSH ..................................... 77
MASTER / END MASTER .............. 80
ORDERED / END ORDERED ......... 81
PARALLEL / END PARALLEL .... 83
PARALLEL DO / END PARALLEL DO 86
PARALLEL SECTIONS / END PARALLEL SECTIONS .................................. 90
PARALLEL WORKSHARE / END PARALLEL WORKSHARE .............................. 93
SCHEDULE ................................ 94
SECTIONS / END SECTIONS ........ 97
SINGLE / END SINGLE .............. 100
TASK ....................................... 104
TASKWAIT ................................ 105
THREAD_LOCAL .......................... 106
THREADPRIVATE ......................... 108
WORKSHARE ................................ 113
OpenMP directive clauses ............ 115
Global rules for directive clauses .... 116
COLLAPSE ................................ 117
COPY ....................................... 119
COPYPRIVATE ............................. 120
DEFAULT ................................. 121
IF ........................................... 123
FIRSTPRIVATE ............................ 124
LASTPRIVATE ............................. 125
NUM_THREADS ............................ 127
ORDERED ................................ 127
PRIVATE .................................. 128
REDUCTION ............................... 129
SCHEDULE .............................. 132
SHARED ................................... 134
UNTIED .................................... 136
OpenMP execution environment, lock and timing routines (IBM extension) .... 136
omp_destroy_lock(ivar) .............. 138
omp_destroy_nest_lock(nvar) ..... 138
omp_get_dynamic() .................... 139
omp_get_max_threads() ............. 139
omp_get_nested() ...................... 140
omp_get_num_procs() ............... 140
omp_get_num_threads() ............. 141
omp_get_thread_num() .............. 142
omp_get_wtick() ...................... 143
omp_get_wtime() ...................... 143
omp_in_parallel() .................... 144
omp_init_lock(svar) ................. 145
omp_init_nest_lock(nvar) ......... 146
omp_set_dynamic(enable_expr) .... 147
omp_set_lock(svar) ................... 147
omp_set_nested(enable_expr) ...... 148
omp_set_nest_lock(nvar) .......... 149
omp_set_num_threads(number_of_threads_expr) 150
omp_test_lock(svar) ................. 151
omp_test_nest_lock(nvar) ........ 151
omp_unset_lock(svar) .............. 152
omp_unset_nest_lock(nvar) ........ 153
Pthreads data structures, functions, and subroutines ............................. 154
f_makefile(delay) ...................... 157
f_pthread_attr_destroy(attr) ...... 157
f_pthread_attr_getdetachstate(attr, detach) ............. 158
f_pthread_attr_getguardsize(attr, guardsize) .......... 159
f_pthread_attr_getinheritsched(attr, inherit) .......... 159
f_pthread_attr_getschedparam(attr, param) .......... 160
f_pthread_attr_getschedpolicy(attr, policy) .......... 160
f_pthread_attr_getscope(attr, scope) ................... 161
f_pthread_attr_getstackaddr(attr, stackaddr) ........ 162
f_pthread_attr_getstacksize(attr, ssize) ............. 162
f_pthread_attr_init(attr) ........... 163
f_pthread_attr_setdetachstate(attr, detach) .......... 164
f_pthread_attr_setguardsize(attr, guardsize) ....... 164
f_pthread_attr_setschedparam(attr, param) .......... 165
f_pthread_attr_setschedpolicy(attr, policy) .......... 166
f_pthread_attr_setscope(attr, scope) ................... 167
f_pthread_attr_setstackaddr(attr, stackaddr) ........ 168
f_pthread_attr_setstacksize(attr, ssize) ............. 168
f_pthread_attr_t ....................... 169
f_pthread_cancel(thread) ........... 169
f_pthread_cleanup_pop(exec) ...... 170
f_pthread_cleanup_push(cleanups, flag, arg) .......... 170
f_pthread_cond_broadcast(cond) .... 172
f_pthread_cond_destroy(cond) ...... 172
f_pthread_cond_init(cond, cattr) .... 173
f_pthread_cond_signal(cond) ...... 173
f_pthread_cond_t ...................... 174
f_pthread_cond_timedwait(cond, mutex, timeout) ...... 174
f_pthread_cond_wait(cond, mutex) .... 175
f_pthread_condattr_destroy(cattr) .... 176
f_pthread_condattr_getshared(cattr, pshared) ........ 176
f_pthread_condattr_init(cattr) ...... 177
f_pthread_condattr_setshared(cattr, pshared) ........ 177
f_pthread_getattr画像(mer, detach) .... 178
f_pthread_getlockparam(thread, policy, param) .... 182
f_pthread_getschedparam(key, arg) ............. 183
f_pthread_join(thread, ret) ....... 183
f_pthread_key_create(key, dir) ...... 184
f_pthread_key_delete(key) ........... 185
f_pthread_key_t ....................... 185
f_pthread_kill(thread, sig) ....... 185
f_pthread_mutex_destroy(mutex) .... 186
f_pthread_mutex_getprioceiling(mutex, old) .......... 187
f_pthread_mutex_init(mutex, mattr) .... 187
f_pthread_mutex_lock(mutex) ....... 188
f_pthread_mutex_setprioceiling(mutex, new, old) .... 188
f_pthread_mutex_t ..................... 189
f_pthread_mutex_trylock(mutex) ...... 189
f_pthread_mutex_unlock(mutex) ...... 190
f_pthread_mutexattr_destroy(mattr) ...... 190
f_pthread_mutexattr_getprioceiling(mattr, ceiling) .... 191
Chapter 8. Interlanguage calls . . . . 213
Conventions for XL Fortran external names . . . 213
Mixed-language input and output . . . . 214
Mixing Fortran and C++ . . . . 215
Making calls to C functions work . . . . 216
Passing data from one language to another . . . 217
Passing arguments between languages . . . 217
Passing global variables between languages . . 218
Passing character types between languages . . . 219
Passing arrays between languages . . . . 220
Passing pointers between languages . . . . 221
Passing arguments by reference or by value . . 221
Returning values from Fortran functions . . . 222
Arguments with the OPTIONAL attribute . . . 223
Type encoding and checking . . . . 223
Assembler-level subroutine linkage conventions . . . 224
The stack . . . . 225
The Link Area . . . . 226
The input parameter area . . . . 227
The register save area . . . . 228
The local stack area . . . . 228
The output parameter area . . . . 228
Linkage convention for argument passing . . . . 229
Argument passing rules (by value) . . . . 230
Order of arguments in argument list . . 232
Linkage convention for function calls . . . . 232
Pointers to functions . . . . 233
Function values . . . . 234
The Stack floor . . . . 234
Stack overflow . . . . 234
Prolog and epilog . . . . 235
Traceback . . . . 235
Stack Size limit . . . . 235
THREADLOCAL common blocks and
interlanguage calls with C . . . 235

Chapter 9. Implementation details of
XL Fortran Input/Output (I/O). . . . . 237
Implementation details of file formats . . 237
File names . . . . 238
Preconnected and Implicitly Connected Files . . 239
File positioning . . . . 239
Preserving the XL Fortran Version 2.3 file
positioning . . . . 240
I/O Redirection . . . . 240
How XL Fortran I/O interacts with pipes, special
files, and links . . . . 241
Default record lengths . . . . 242
File permissions . . . . 242
Selecting error messages and recovery actions . . 242
Flushing I/O buffers . . . . 243
Choosing locations and names for Input/Output
files . . . . 243
Naming files that are connected with no explicit
name . . . . 244
Naming scratch files . . . . 244
Increasing throughput with logical volume I/O
and data striping . . . . 245
Logical volume I/O . . . . 245
Data striping . . . . 246
Asynchronous I/O . . . . 246
Execution of an asynchronous data transfer
operation . . . . 247
Usage . . . . 247
Performance . . . . 249
Compiler-generated temporary I/O items . . 250
System setup . . . . 250
Linking . . . . 251
Error handling . . . . 252
XL Fortran thread-safe I/O library . . . . 252
Use of I/O statements in signal handlers . . . 255
Asynchronous thread cancellation . . . . 256

Chapter 10. Implementation details of
XL Fortran floating-point processing . . . . 257
IEEE Floating-point overview . . . . 257
Compiling for strict IEEE conformance . . . . 257
IEEE Single- and double-precision values . . 258
IEEE Extended-precision values . . . . 258
Infinities and NaNs . . . . 258
Exception-handling model . . . . 259
Hardware-specific floating-point overview . . . . 260
Single- and double-precision values 260
Extended-precision values 261
How XL Fortran rounds floating-point calculations 262
Selecting the rounding mode 262
Minimizing rounding errors 264
Minimizing overall rounding 264
Delaying rounding until run time 264
Ensuring that the rounding mode is consistent 264
Duplicating the floating-point results of other systems 265
Maximizing floating-point performance 265
Detecting and trapping floating-point exceptions 267
Compiler features for trapping floating-point exceptions 267
Operating system features for trapping floating-point exceptions 268
Installing an exception handler 268
Producing a core file 269
Controlling the floating-point status and control register 269
xlf_fp_util Procedures 270
fpgets and fsets subroutines 271
Sample programs for exception handling 272
Causing exceptions for particular variables 272
Minimizing the performance impact of floating-point exception trapping 273
Floating-point processing on the POWER and POWER2 architectures 273
Precision of computations 274
Invalid operation exceptions for SQRT operations on POWER processors 274

Chapter 11. Porting programs to XL
Fortran 277
Outline of the porting process 277
Maintaining FORTRAN 77 source and object code 277
Portability of directives 277
Common industry extensions that XL Fortran supports 278
Mixing data types in statements 280
Date and time routines 280
Other libc routines 280
Changing the default sizes of data types 280
Name conflicts between your procedures and XL Fortran intrinsic procedures 280
Reproducing results from other systems 280
Finding nonstandard extensions 280

Chapter 12. Sample Fortran programs 283
Example 1 - XL Fortran source file 283
Example 2 - valid C routine source file 283
Example 3 - valid Fortran SMP source file 286
Example 4 - invalid Fortran SMP source file 286
Programming examples using the Pthreads library module 287

Notices 289
Trademarks and service marks 291

Index 293
About this information

This information is part of the IBM® XL Fortran for AIX®, V12.1 information suite. It provides both reference information and practical tips for using XL Fortran’s optimization and tuning capabilities to maximize application performance, as well as expanding on programming concepts such as I/O and interlanguage calls.

Who should read this information

This information is for anyone who wants to exploit the XL Fortran compiler’s capabilities for optimizing and tuning Fortran programs. Readers should be familiar with their AIX operating system and have extensive Fortran programming experience with complex applications. However, users new to XL Fortran can still use this information to help them understand how the compiler’s features can be used for effective program optimization.

How to use this information

This guide focuses on specific programming and compilation techniques that can maximize XL Fortran application performance. It covers optimization and tuning strategies, recommended programming practices and compilation procedures, debugging, and information on using XL Fortran advanced language features. This guide also contains cross-references to relevant topics of other reference guides in the XL Fortran information suite.

Topics not described in this information are available as indicated in the following:

- Installation, system requirements, last-minute updates: see the XL Fortran Installation Guide and product README.
- Overview of XL Fortran features: see the Getting Started with XL Fortran.
- Syntax, semantics, and implementation of the XL Fortran programming language: see the XL Fortran Language Reference.
- Compiler setup, compiling and running programs, compiler options, diagnostics: see the XL Fortran Compiler Reference.
- Operating system commands related to the use of the compiler: AIX Commands Reference, Volumes 1 - 6 and the AIX information center.

How this information is organized

This guide includes the following topics:

- Chapter 1, “Optimizing your applications,” on page 1 provides an overview of the optimization process.
- Chapter 2, “Tuning XL compiler applications,” on page 21 discusses the compiler options available for optimizing and tuning code.
- Chapter 3, “Advanced optimization concepts,” on page 33, Chapter 4, “Managing code size,” on page 37, and “Debugging optimized code” on page 15 discuss advanced techniques like optimizing loops and inlining code, and debug considerations for optimized code.
• The following sections contain information on how to write optimization
friendly, portable XL Fortran code, that is interoperable with other languages.
Also included is a description of XL Fortran’s OpenMP and SMP support with
guidelines for writing parallel code.
  – Chapter 5, “Compiler-friendly programming techniques,” on page 43
  – Chapter 6, “High performance libraries,” on page 45
  – Chapter 7, “Parallel programming with XL Fortran,” on page 57
  – Chapter 8, “Interlanguage calls,” on page 213

• The following sections contain information about XL Fortran and its
implementation that can be useful for new and experienced users alike, as well
as those who want to move their existing Fortran applications to the XL Fortran
compiler:
  – Chapter 9, “Implementation details of XL Fortran Input/Output (I/O),” on
    page 237
  – Chapter 10, “Implementation details of XL Fortran floating-point processing,”
    on page 257
  – Chapter 11, “Porting programs to XL Fortran,” on page 277

**Conventions**

**Typographical conventions**

The following table explains the typographical conventions used in the IBM XL
Fortran for AIX, V12.1 information.

*Table 1. Typographical conventions*

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

**Qualifying elements (icons and bracket separators)**

In descriptions of language elements, this information uses marked bracket
separators to delineate large blocks of text and icons to delineate small segments of
text as follows:
Table 2. Qualifying elements

<table>
<thead>
<tr>
<th>Bracket separator text</th>
<th>Icon</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran 95 begins / ends</td>
<td>![F95 icon]</td>
<td>The text describes an IBM XL Fortran implementation of the Fortran 95 standard.</td>
</tr>
<tr>
<td>IBM extension begins / ends</td>
<td>![IBM icon]</td>
<td>The text describes a feature that is an IBM XL Fortran extension to the standard language specifications.</td>
</tr>
</tbody>
</table>

Syntax diagrams

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

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.
  - The ⟷— symbol indicates the beginning of a command, directive, or statement.
  - The ⟷—- symbol indicates that the command, directive, or statement syntax is continued on the next line.
  - The —— symbol indicates that a command, directive, or statement is continued from the previous line.
  - The —— symbol indicates the end of a command, directive, or statement.
- Fragments, which are diagrams of syntactical units other than complete commands, directives, or statements, start with the | —— symbol and end with the —— symbol.
  - IBM XL Fortran extensions are marked by a number in the syntax diagram with an explanatory note immediately following the diagram.
  - Program units, procedures, constructs, interface blocks and derived-type definitions consist of several individual statements. For such items, a box encloses the syntax representation, and individual syntax diagrams show the required order for the equivalent Fortran statements.
- Required items are shown on the horizontal line (the main path):
  - ⟷—keyword ——required_argument— ——
- Optional items are shown below the main path:
  - ⟷—keyword ——optional_argument

Note: Optional items (not in syntax diagrams) are enclosed by square brackets ([ and ]). For example, [UNIT=*]u
- If you can choose from two or more items, they are shown vertically, in a stack.
If you must choose one of the items, one item of the stack is shown on the main path.

```plaintext
>>> keyword
     required_argument1
     required_argument2
```

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

```plaintext
>>> keyword
     optional_argument1
     optional_argument2
```

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

```plaintext
>>> keyword
     repeatable_argument
```

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

```plaintext
>>> keyword
     default_argument
     alternate_argument
```

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

### Sample syntax diagram

The following is an example of a syntax diagram with an interpretation:
How to read syntax statements

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

Example of a syntax statement

```
EXAMPLE char_constant {a|b}[c|d][e|,... 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:
1. IBM extension

Interpret the diagram as follows:
- Enter the keyword EXAMPLE.
- EXAMPLE is an IBM extension.
- 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.
- 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. (The _list syntax is equivalent to the previous syntax for `e`.)
Examples in this information

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

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

Notes on the terminology used

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

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

Related information

The following sections provide related information for XL Fortran:

IBM XL Fortran information

XL Fortran provides product information in the following formats:

- README files
  README files contain late-breaking information, including changes and corrections to the product information. README files are located by default in the XL Fortran directory and in the root directory of the installation CD.
- Installable man pages
  Man pages are provided for the compiler invocations and all command-line utilities provided with the product. Instructions for installing and accessing the man pages are provided in the IBM XL Fortran for AIX, V12.1 Installation Guide.
- Information center
  The information center of searchable HTML files can be launched on a network and accessed remotely or locally. Instructions for installing and accessing the online information center are provided in the IBM XL Fortran for AIX, V12.1 Installation Guide.

  The information center is viewable on the Web at [http://publib.boulder.ibm.com/infocenter/comphelp/v101v121/index.jsp](http://publib.boulder.ibm.com/infocenter/comphelp/v101v121/index.jsp)
- PDF documents
  PDF documents are located by default in the /usr/lpp/xlf/doc/LANG/pdf directory, where LANG is one of en_US or ja_JP. The PDF files are also available on the Web at [http://www.ibm.com/software/awdtools/fortran/xlforntrang/library](http://www.ibm.com/software/awdtools/fortran/xlforntrang/library)
The following files comprise the full set of XL Fortran product information:

**Table 3. XL Fortran PDF files**

<table>
<thead>
<tr>
<th>Document title</th>
<th>PDF file name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM XL Fortran for AIX, V12.1 Installation Guide, GC23-8893-00</td>
<td>install.pdf</td>
<td>Contains information for installing XL Fortran and configuring your environment for basic compilation and program execution.</td>
</tr>
<tr>
<td>Getting Started with IBM XL Fortran for AIX, V12.1, GC23-8898-00</td>
<td>getstart.pdf</td>
<td>Contains an introduction to the XL Fortran product, with information on setting up and configuring your environment, compiling and linking programs, and troubleshooting compilation errors.</td>
</tr>
<tr>
<td>IBM XL Fortran for AIX, V12.1 Compiler Reference, SC23-8892-00</td>
<td>compiler.pdf</td>
<td>Contains information about the various compiler options and environment variables.</td>
</tr>
<tr>
<td>IBM XL Fortran for AIX, V12.1 Language Reference, SC23-8891-00</td>
<td>langref.pdf</td>
<td>Contains information about the Fortran programming language as supported by IBM, including language extensions for portability and conformance to nonproprietary standards, compiler directives and intrinsic procedures.</td>
</tr>
<tr>
<td>IBM XL Fortran for AIX, V12.1 Optimization and Programming Guide, SC23-8894-00</td>
<td>proguide.pdf</td>
<td>Contains information on advanced programming topics, such as application porting, interlanguage calls, floating-point operations, input/output, application optimization and parallelization, and the XL Fortran high-performance libraries.</td>
</tr>
</tbody>
</table>

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

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


**Standards and specifications**

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

- Federal (USA) Information Processing Standards Publication Fortran, FIPS PUB 69-1.
- Information technology - Programming languages - Fortran - Part 1: Base language, ISO/IEC 1539-1:1997. (This information uses its informal name, Fortran 95.)
- Information technology - Programming languages - Fortran - Part 1: Base language, ISO/IEC 1539-1:2004. (This information uses its informal name, Fortran 2003.)
- Information technology - Programming languages - Fortran - Enhanced data type facilities, ISO/IEC JTC1/SC22/WG5 N1379.
Information technology - Programming languages - Fortran - Floating-point exception handling, 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 Defense standard). Note that XL Fortran supports only those extensions documented in this standard that have also been subsequently incorporated into the Fortran 90 standard.


Other IBM information

- AIX Commands Reference, Volumes 1 - 6, SC23-4888
- AIX General Programming Concepts: Writing and Debugging Programs, SC23-4896
- AIX Assembler Language Reference, SC23-4923

All AIX information is available at http://publib.boulder.ibm.com/infocenter/pseries/v5r3/index.jsp

- Parallel Environment for AIX: Operation and Use


Technical support

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

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

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

How to send your comments

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

Be sure to include the name of the information, the part number of the information, the version of XL Fortran, and, if applicable, the specific location of the text you are commenting on (for example, a page number or table number).
Chapter 1. Optimizing your applications

The XL compilers enable development of high performance 32-bit and 64-bit applications by offering a comprehensive set of performance enhancing techniques that exploit the multilayered PowerPC® architecture. These performance advantages depend on good programming techniques, thorough testing and debugging, followed by optimization, and tuning.

Distinguishing between optimization and tuning

You can use optimization and tuning separately or in combination to increase the performance of your application. Understanding the difference between them is the first step in understanding how the different levels, settings and techniques can increase performance.

Optimization

Optimization is a compiler driven process that searches for opportunities to restructure your source code and give your application better overall performance at runtime, without significantly impacting development time. The XL compiler optimization suite, which you control using compiler options and directives, performs best on well-written source code that has already been through a thorough debugging and testing process. These optimization transformations can:

- Reduce the number of instructions your application executes to perform critical operations.
- Restructure your object code to make optimal use of the PowerPC architecture.
- Improve memory subsystem usage.
- Exploit the ability of the architecture to handle large amounts of shared memory parallelization.

Consider that although not all optimizations benefit all applications, even basic optimization techniques can result in a performance benefit. Consult the Steps in the optimization process for an overview of the common sequence of steps you can use to increase the performance of your application.

Tuning

Where optimization applies increasingly aggressive transformations designed to improve the performance of any application in any supported environment, tuning offers you opportunities to adjust characteristics of your application to improve performance, or to target specific execution environments. Even at low optimization levels, tuning for your application and target architecture can have a positive impact on performance. With proper tuning the compiler can:

- Select more efficient machine instructions.
- Generate instruction sequences that are more relevant to your application.

For instructions, see Tuning XL compiler applications.
Steps in the optimization process

As you begin the optimization process, consider that not all optimization techniques suit all applications. Trade-offs sometimes occur between an increase in compile time, a reduction in debugging capability, and the improvements that optimization can provide.

Learning about, and experimenting with different optimization techniques can help you strike the right balance for your XL compiler applications while achieving the best possible performance. Also, though it is unnecessary to hand-optimize your code, compiler-friendly programming can be extremely beneficial to the optimization process. Unusual constructs can obscure the characteristics of your application and make performance optimization difficult. Use the steps in this section as a guide for optimizing your application.

1. The Basic optimization step begins your optimization processes at levels 0 and 2.
2. The Advanced optimization step exposes your application to more intense optimizations at levels 3 through 5.
3. The High-order transformation (HOT) step can help you limit loop execution time.
4. The Interprocedural analysis (IPA) step can optimize your entire application at once.
5. The Profile-directed feedback (PDF) step focuses optimizations on specific characteristics of your application.
6. The Debugging high-performance code step can help you identify issues and problems that can occur with optimized code.
7. The Getting more performance section offers other strategies and tuning alternatives to compiler-driven optimization.

The section Compiler-friendly programming techniques contains tips for writing more easily optimized source code.

Basic optimization

The XL compiler supports several levels of optimization, with each option level building on the levels below through increasingly aggressive transformations, and consequently using more machine resources.

Ensure that your application compiles and executes properly at low optimization levels before trying more aggressive optimizations. This topic discusses two optimizations levels, listed with complementary options in the Basic optimizations table. The table also includes a column for compiler options that can have a performance benefit at that optimization level for some applications.

<table>
<thead>
<tr>
<th>Optimization level</th>
<th>Additional options implied by default</th>
<th>Complementary options</th>
<th>Other options with possible benefits</th>
</tr>
</thead>
<tbody>
<tr>
<td>-O0</td>
<td>None</td>
<td>-qarch</td>
<td>-g</td>
</tr>
<tr>
<td>-O2</td>
<td>-qmaxmem=8192</td>
<td>-qarch -qtune</td>
<td>-qmaxmem=-1 -qhot=level=0</td>
</tr>
</tbody>
</table>

Note: Specifying -O without including a level implies -O2.
Optimizing at level 0

Benefits at level 0
- Minimal performance improvement, with minimal impact on machine resources.
- Exposes some source code problems, helping in the debugging process.

Begin your optimization process at -O0 which the compiler already specifies by default. For SMP programs, the closest equivalent to -O0 is -qsmp=noopt. This level performs basic analytical optimization by removing obviously redundant code, and can result in better compile time, while ensuring your code is algorithmically correct so you can move forward to more complex optimizations.

-O0 also includes constant folding. The option -qfloat=nofold can be used to suppress folding floating-point operations. Optimizing at this level accurately preserves all debug information and can expose problems in existing code, such as uninitialized variables.

Additionally, specifying -qarch at this level targets your application for a particular machine and can significantly improve performance by ensuring your application takes advantage of all applicable architectural benefits.

For more information on tuning, consult [Tuning for Your Target Architecture](#).

See the [O option](#) in the XL Fortran Compiler Reference for information on the -O level syntax.

Optimizing at level 2

Benefits at level 2
- Eliminates redundant code
- Basic loop optimization
- Can structure code to take advantage of -qarch and -qtune settings

After successfully compiling, executing, and debugging your application using -O0, recompiling at -O2 opens your application to a set of comprehensive low-level transformations that apply to subprogram or compilation unit scopes and can include some inlining. Optimizations at -O2 are a relative balance between increasing performance while limiting the impact on compilation time and system resources. You can increase the memory available to some of the optimizations in the -O2 portfolio by providing a larger value for the -qmaxmem option. Specifying -qmaxmem=-1 allows the optimizer to use memory as needed without checking for limits but does not change the transformations the optimizer applies to your application at -O2.

Starting to tune at level 2

Choosing the right hardware architecture target or family of targets becomes even more important at -O2 and higher. Targeting the proper hardware allows the optimizer to make the best use of the hardware facilities available. If you choose a family of hardware targets, the -qtune option can direct the compiler to emit code consistent with the architecture choice, but will execute optimally on the chosen tuning hardware target. This allows you to compile for a general set of targets but have the code run best on a particular target.

See the [Tuning for Your Target Architecture](#) topics for details on the -qarch and -qtune options.
The -O2 option can perform a number of additional optimizations, including:

- Common subexpression elimination: Eliminates redundant instructions.
- Constant propagation: Evaluates constant expressions at compile-time.
- Dead code elimination: Eliminates instructions that a particular control flow does not reach, or that generate an unused result.
- Dead store elimination: Eliminates unnecessary variable assignments.
- Graph coloring register allocation: Globally assigns user variables to registers.
- Value numbering: Simplifies algebraic expressions, by eliminating redundant computations.
- Instruction scheduling for the target machine.
- Loop unrolling and software pipelining.
- Moves invariant code out of loops.
- Simplifies control flow.
- Strength reduction and effective use of addressing modes.

Even with -O2 optimizations, some useful information about your source code is made available to the debugger if you specify -g. Conversely, higher optimization levels can transform code to an extent to which debug information is no longer accurate. Use that information with discretion.

The section on Debugging Optimized Code discusses other debugging strategies in detail.

See the -O option in the XL Fortran Compiler Reference for information on the -O level syntax.

**Advanced optimization**

Higher optimization levels can have a tremendous impact on performance, but some trade-offs can occur in terms of code size, compilation time, resource requirements, and numeric or algorithmic precision.

After applying basic optimizations and successfully compiling and executing your application, you can apply more powerful optimization tools. The XL compiler optimization portfolio includes many options for directing advanced optimization, and the transformations your application undergoes are largely under your control. The discussion of each optimization level in Table 5 includes information on not only the performance benefits, and the possible trade-offs as well, but information on how you can help guide the optimizer to find the best solutions for your application.

**Table 5. Advanced optimizations**

<table>
<thead>
<tr>
<th>Optimization Level</th>
<th>Additional options implied</th>
<th>Complementary options</th>
<th>Options with possible benefits</th>
</tr>
</thead>
<tbody>
<tr>
<td>-O3</td>
<td>-qnostrict</td>
<td>-qarch</td>
<td>-qpdf</td>
</tr>
<tr>
<td></td>
<td>-qmaxmem=-1</td>
<td>-qtune</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-qhot=level=0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 5. Advanced optimizations (continued)

<table>
<thead>
<tr>
<th>-O4</th>
<th>-nostrict</th>
<th>-qmaxmem=-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>-qhot</td>
<td>-qarch=auto</td>
<td>-qcache</td>
</tr>
<tr>
<td>-qipa</td>
<td>-qcache=auto</td>
<td></td>
</tr>
<tr>
<td>-qtune=auto</td>
<td>-qcache=auto</td>
<td></td>
</tr>
<tr>
<td>-qipa=level=2</td>
<td>-qarch</td>
<td>-qtune</td>
</tr>
<tr>
<td>-qcache</td>
<td>-qarch=auto</td>
<td>-qtune</td>
</tr>
<tr>
<td>-qipa=level=2</td>
<td>-qcache</td>
<td>-qtune</td>
</tr>
<tr>
<td>-qcache</td>
<td>-qarch=auto</td>
<td>-qtune</td>
</tr>
</tbody>
</table>

Optimizing at level 3

Benefits at level 3

- In-depth aliasing analysis
- Better loop scheduling
- High-order loop analysis and transformations (-qhot=level=0)
- Inlining of small procedures within a compilation unit by default
- Eliminating implicit compile-time memory usage limits
- Widening, which merges adjacent load/stores and other operations
- Pointer aliasing improvements to enhance other optimizations

Specifying -O3 initiates more intense low-level transformations that remove many of the limitations present at -O2. For instance, the optimizer no longer checks for memory limits, by defaulting to -qmaxmem=-1. Additionally, optimizations encompass larger program regions and attempt more in-depth analysis. While not all applications contain opportunities for the optimizer to provide a measurable increase in performance, most applications can benefit from this type of analysis.

Potential trade-offs at level 3

With the in-depth analysis of -O3 comes a trade-off in terms of compilation time and memory resources. Also, since -O3 implies -nostrict, the optimizer can alter certain floating-point semantics in your application to gain execution speed. This typically involves precision trade-offs as follows:

- Reordering of floating-point computations.
- Reordering or elimination of possible exceptions, such as division by zero or overflow.

You can still gain most of the -O3 benefits while preserving precise floating-point semantics by specifying -strict. Compiling with -strict is necessary if you require the same absolute precision in floating-point computational accuracy as you get with -O0, -O2, or -noopt results. The option -strict=ieee also ensures adherence to all IEEE semantics for floating-point operations. If your application is sensitive to floating-point exceptions or the order of evaluation for floating-point arithmetic, compiling with -strict, -strict=exceptions, or -strict=order, helps to ensure accurate results. You should also consider the impact of the -strict=precision suboption group on floating-point computational accuracy. The precision suboption group includes the individual suboptions: subnormals, operationprecision, association, reductionorder, and library (described in the -strict option in the XL Fortran Compiler Reference).
Without `-qstrict`, the difference in computation for any one source-level operation is very small in comparison to basic optimization. Although a small difference can be compounded if the operation is in a loop structure where the difference becomes additive, most applications are not sensitive to the changes that can occur in floating-point semantics.

See the `-O` option in the XL Fortran Compiler Reference for information on the `-O` level syntax.

**An intermediate step: adding `-qhot` suboptions at level 3**

At `-O3`, the optimization includes minimal `-qhot` loop transformations at level=0 to increase performance. You can further increase your performance benefit by increasing the level and therefore the aggressiveness of `-qhot`. Try specifying `-qhot` without any suboptions, or `-qhot=level=1`.

The following `-qhot` suboptions can also provide additional performance benefits, depending on the characteristics of your application:

- `-qhot=simd` to enable short vectorization
- `-qhot=vector` to enable long vectorization
- `-qhot=arrypad` to enable array padding

For more information on `-qhot`, see [High-order transformation (HOT)](https://www.ibm.com/support/knowledgecenter/SSLTBW_AIX_71/fortran/comp_xlf_optimizing.html#ixref/comp_xlf_optimizing/-qhot).

**Optimizing at level 4**

**Benefits at level 4**

- Propagation of global and argument values between compilation units
- Inlining code from one compilation unit to another
- Reorganization or elimination of global data structures
- An increase in the precision of aliasing analysis

Optimizing at `-O4` builds on `-O3` by triggering `-qipa=level=1` which performs interprocedural analysis (IPA), optimizing your entire application as a unit. This option is particularly pertinent to applications that contain a large number of frequently used routines.

To make full use of IPA optimizations, you must specify `-O4` on the compilation and link steps of your application build as interprocedural analysis occurs in stages at both compile and link time.

**Potential trade-offs at level 4**

In addition to the trade-offs already mentioned for `-O3`, specifying `-qipa` can significantly increase compilation time, especially at the link step.

See the `-O` option in the XL Fortran Compiler Reference for information on the `-O` level syntax.

**The IPA process**

1. At compilation time optimizations occur on a file-by-file basis, as well as preparation for the link stage. IPA writes analysis information directly into the object files the compiler produces.
2. At the link stage, IPA reads the information from the object files and analyzes the entire application.
3. This analysis guides the optimizer on how to rewrite and restructure your application and apply appropriate -O3 level optimizations.

The “Specialized optimization techniques” on page 8 section contains more information on IPA including details on IPA suboptions.

Beyond -qipa, -O4 enables other optimization options:

- **-qhot**
  Enables more aggressive HOT transformations to optimize loop constructs and array language.

- **-qhot=vector**
  Optimizes array data to run mathematical operations in parallel where applicable.

- **-qarch=auto and -qtune=auto**
  Optimizes your application to execute on a hardware architecture identical to your build machine. If the architecture of your build machine is incompatible with your application’s execution environment, you must specify a different -qarch suboption after the -O4 option. This overrides -qarch=auto.

- **-qcache=auto**
  Optimizes your cache configuration for execution on specific hardware architecture. The auto suboption assumes that the cache configuration of your build machine is identical to the configuration of your execution architecture. Specifying a cache configuration can increase program performance, particularly loop operations by blocking them to process only the amount of data that can fit into the data cache.
  If you will be executing your application on a different machine, specify correct cache values.

### Optimizing at level 5

#### Benefits at level 5

- Most aggressive optimizations available
- Makes full use of loop optimizations and “Specialized optimization techniques” on page 8

As the highest optimization level, -O5 includes all -O4 optimizations and deepens whole program analysis by increasing the -qipa level to 2. Compiling with -O5 also increases how aggressively the optimizer pursues aliasing improvements. Additionally, if your application contains a mix of C/C++ and Fortran code that you compile using XL compilers, you can increase performance by compiling and linking your code with the -O5 option.

#### Potential trade-offs at level 5

Compiling at -O5 requires more compilation time and machine resources than any other optimization level, particularly if you include -O5 on the IPA link step. Compile at -O5 as the final phase in your optimization process after successfully compiling and executing your application at -O4.

See the -O option in the XL Fortran Compiler Reference for information on the -O level syntax.
Specialized optimization techniques

While some optimization techniques are active at advanced optimization levels, certain types of applications can receive a performance benefit even when you apply only basic optimizations.

Table 6. Specialized optimization techniques

<table>
<thead>
<tr>
<th>Technique</th>
<th>Benefit</th>
</tr>
</thead>
<tbody>
<tr>
<td>HOT</td>
<td>Minimizes loop execution time which is beneficial to most applications that contain large loops, or many small loops. HOT also improves memory access patterns in your application.</td>
</tr>
<tr>
<td>IPA</td>
<td>Performs whole program analysis, providing the optimization suite with a complete view of your entire application. This applies performance enhancements with more focus and robustness.</td>
</tr>
<tr>
<td>PDF</td>
<td>Targets the code paths your application executes most frequently for optimization.</td>
</tr>
</tbody>
</table>

High-order transformation (HOT)

As part of the XL compiler optimization suite, the HOT transformations focus specifically on loops which typically account for the majority of the execution time for most applications. HOT transformations perform in-depth loop analysis to minimize their execution time. Loop optimization analysis includes:

- Interchange
- Fusion
- Unrolling loop nests
- Reducing the use of temporary arrays

The goals of these optimizations include:

- Reducing memory access costs through effective cache use and translation look-aside buffers (TLBs). Increasing memory locality reduces cache and TLB misses.
- Overlapping computation and memory access through effective utilization of the hardware data prefetching capabilities.
- Improving processor resource utilization by reordering and balancing the use of instructions with complementary resource requirements. Loop computation balance typically involves creating an equitable relationship between load/store operations and floating-point computations.

Compiling with -O3 and higher triggers HOT transformations by default. You can also see performance benefits by specifying -qhot with -O2, or adding more -qhot optimizations than the default level=0 at -O3.

You can see particular -qhot benefits if your application contains Fortran 90-style array language constructs, as HOT transformations include elimination of intermediate temporary variables and statement fusion.

You can also use directives to assist in loop analysis. Assertive directives such as INDEPENDENT or CNCALL allow you to describe important loop characteristics.
or behaviors that HOT transformations can exploit. Prescriptive directives such as UNROLL or PREFETCH allow you to direct the HOT transformations on a loop-by-loop basis. You can also specify the -qreport compiler option to generate information about loop transformations. The report can assist you in deciding where best to include directives to improve the performance of your application. For example, you can use this section of the listing to identify non-stride-one references that may prevent loop vectorization.

In addition to general loop transformation, -qhot supports suboptions that you can specify to enable additional transformations detailed in this section.

**HOT short vectorization**

When targeting a PowerPC processor that supports Vector Multimedia Extension (VMX), specifying -qhot=simd allows the optimizer to transform code into VMX instructions when you are compiling with -qenablevmx. These machine instructions can execute up to sixteen operations in parallel. The most common opportunity for this transformation is with loops that iterate over contiguous array data, performing calculations on each element. You can use the NOSIMD directive to prevent the transformation of a particular loop.

**HOT long vectorization**

When you specify any of the following:
- -O4 and higher
- -O3 with -qhot=level=1 and -qnostrict
- -qhot with -qnostrict

you enable -qhot=vector by default. Specifying -qnostrict with optimizations other than -O4 and -O5 ensures that the compiler looks for long vectorization opportunities. This can optimize loops in source code for operations on array data by ensuring that operations run in parallel where applicable. The compiler uses standard machine registers for these transformations and does not restrict vector data size; supporting both single- and double-precision floating-point vectorization. Often, HOT vectorization involves transformations of loop calculations into calls to specialized mathematical routines supplied with the compiler such as the Mathematical Acceleration Subsystem (MASS) libraries. These mathematical routines use algorithms that calculate results more efficiently than executing the original loop code.

For more information on optimization levels like -O4 and the other compiler options they imply, see "Advanced optimization” on page 4.

**HOT array size adjustment**

An array dimension that is a power of two can lead to a decrease in cache utilization. The -qhot=arrraypad suboption allows the compiler to increase the dimensions of arrays where doing so could improve the efficiency of array-processing loops. Using this suboption can reduce cache misses and page faults that slow your array processing programs. The HOT transformations will not necessarily pad all arrays, and can pad different arrays by different amounts in order to gain performance. You can specify a padding factor to apply to all arrays. This value is typically a multiple of the largest array element size.

Pad arrays with discretion as array padding uses more memory and the performance trade-off does not benefit all applications. Also, these HOT
transformations do not include checks for array data overlay, as with Fortran EQUIVALENCE, or array shaping operations.

**Interprocedural analysis (IPA)**

Interprocedural Analysis (IPA) can analyze and optimize your application as a whole, rather than on a file-by-file basis. Run during the link step of an application build, the entire application, including linked libraries, is available for interprocedural analysis. This whole program analysis opens your application to a powerful set of transformations available only when more than one file or compilation unit is accessible. IPA optimizations are also effective on mixed language applications.

![Diagram of IPA process]

*Figure 1. IPA at the link step*

The following are some of the link-time transformations that IPA can use to restructure and optimize your application:

- Inlining between compilation units
- Complex data flow analyses across subprogram calls to eliminate parameters or propagate constants directly into called subprograms.
- Improving parameter usage analysis, or replacing external subprogram calls to system libraries with more efficient inline code.
- Restructuring data structures to maximize access locality.

In order to maximize IPA link-time optimization, you must use IPA at both the compile and link step. Objects you do not compile with IPA can only provide minimal information to the optimizer, and receive minimal benefit. However when IPA is active on the compile step, the resulting object file contains program information that IPA can read during the link step. The program information is invisible to the system linker, and you can still use the object file and link without invoking IPA. The IPA optimizations use hidden information to reconstruct the original compilation and can completely analyze the subprograms the object contains in the context of their actual usage in your application.
During the link step, IPA restructures your application, partitioning it into distinct logical code units. After IPA optimizations are complete, IPA applies the same low-level compilation-unit transformations as the -O2 and -O3 base optimizations levels. Following those transformations, the compiler creates one or more object files and linking occurs with the necessary libraries through the system linker.

It is important that you specify a set of compilation options as consistent as possible when compiling and linking your application. This includes all compiler options, not just -qipa suboptions. When possible, specify identical options on all compilations and repeat the same options on the IPA link step. Incompatible or conflicting options that you specify to create object files, or link-time options in conflict with compile-time options can reduce the effectiveness of IPA optimizations.

**Using IPA on the compile step only**

IPA can still perform transformations if you do not specify IPA on the link step. Using IPA on the compile step initiates optimizations that can improve performance for an individual object file even if you do not link the object file using IPA. The primary focus of IPA is link-step optimization, but using IPA only on the compile-step can still be beneficial to your application without incurring the costs of link-time IPA.

**IPA Levels and other IPA suboptions**

You can control many IPA optimization functions using the -qipa option and suboptions. The most important part of the IPA optimization process is the level at which IPA optimization occurs. Default compilation does not invoke IPA. If you specify -qipa without a level, or specify -O4 IPA optimizations are at level one. If you specify -O5 IPA optimizations are at level two.

<table>
<thead>
<tr>
<th>IPA Level</th>
<th>Behaviors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Figure 2. IPA at the compile step*
IPA includes many suboptions that can help you guide IPA to perform optimizations important to the particular characteristics of your application. Among the most relevant to providing information on your application are:

- **lowfreq** which allows you to specify a list of procedures that are likely to be called infrequently during the course of a typical program run. Performance can increase because optimization transformations will not focus on these procedures.

- **partition** which allows you to specify 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.

- **threads** which allows you to specify the number of parallel threads available to IPA optimizations. This can provide an increase in compilation-time performance on multi-processor systems.

- **clonearch** which allows you to instruct the compiler to generate duplicate subprograms with each tuned to a particular architecture.

**Using IPA across the XL compiler family**

The XL compiler family shares optimization technology. Object files you create using IPA on the compile step with the XL C, C++, and Fortran compilers can undergo IPA analysis during the link step. Where program analysis shows that objects were built with compatible options, such as `-qnostrict`, IPA can perform transformations such as inlining C functions into Fortran code, or propagating C++ constant data into C function calls.

| `qipa=level=0` | - Automatically recognizes standard library functions
|               | - Localizes statically bound variables and procedures
|               | - Organizes and partitions your code according to call affinity, expanding the scope of the `-O2` and `-O3` low-level compilation unit optimizer
|               | - Lowers compilation time in comparison to higher levels, though limits analysis |
| `qipa=level=1` | - Level 0 optimizations
|               | - Performs procedure inlining across compilation units
|               | - Organizes and partitions static data according to reference affinity |
| `qipa=level=2` | - Level 0 and level 1 optimizations
|               | - Performs whole program alias analysis which removes ambiguity between pointer references and calls, while refining call side effect information
|               | - Propagates interprocedural constants
|               | - Eliminates dead code
|               | - Performs pointer analysis
|               | - Performs procedure cloning
|               | - Optimizes intraprocedural operations, using specifically:
|               |   - Value numbering
|               |   - Code propagation and simplification
|               |   - Code motion, into conditions and out of loops
|               |   - Redundancy elimination techniques |

Table 7. The levels of IPA (continued)
Profile-directed feedback (PDF)

Beginning with -O4, compiling with -qpdf to trigger profile-directed feedback is a viable option to increase performance in many applications. Profile-directed feedback is a two-stage compilation process that provides the compiler with the execution path characteristic of your application’s typical behavior after a sample execution. The optimizer uses that information to focus optimization trade-offs in favour of code that executes more frequently.

- PDF at Stage 1: Compiling with [-qpdf1] instruments your code with calls to the PDF runtime library that are linked with your application. After compilation, execute your application with typical input data. You can execute your application with as many data sets as you have, each run records PDF information in data files. Avoid using atypical data which can skew the analysis to favour infrequently executed code paths.

- PDF at Stage 2: After collecting PDF information, recompiling or relinking your application using [-qpdf2] allows the compiler to read information from the PDF data files and makes that information available to the optimizer. Using this data, the optimizer can better direct transformations to facilitate more intense performance gains.

**PDF walkthrough**

The following steps are a guide through PDF optimization. These steps also include the use of utilities designed to enhance the PDF process. Although you can specify PDF optimization (-qpdf) as early in the optimization process as -O2, PDF optimization is recommended at -O4 and higher.

1. Compile your application using -qpdf1 and -qshowpdf.
2. Run your application using one or more characteristic data sets.
3. Use the showpdf utility if you wish to view the information in the PDF file.

To exert more control over the PDF process, use the following steps:

1. Compile your application with -qpdf1.
2. Run your application with one or more characteristic data sets. This produces a PDF file in the current directory.
3. Copy your application to another directory and run it again. This produces a PDF file in the second directory. You can repeat this step multiple times.
4. Use the `mergepdf` utility to combine all PDF files into a single 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 the following syntax:

```bash
mergepdf -r 53 path1 -r 32 path2 -r 15 path3
```

5. Compile the application with `-qpdf2`.

Alternatively, you can use `-qpdf2` to link the object files the `-qpdf1` pass creates without recompiling your source on the `-qpdf2` pass. This alternate approach can save considerable time and help tune large applications for optimization. You can create and test different styles of PDF optimized binaries by specifying different options on the `-qpdf2` pass.

To erase PDF information in a directory, use the `cleanpdf` or `resetpdf` utility.

**Object level profile-directed feedback**

In addition to optimizing entire executables, profile-directed feedback (PDF) can also be applied to specific objects. This can be an advantage in applications where patches or updates are distributed as object files or libraries rather than as executables. Also, specific areas of functionality in your application can be optimized without you needing to go through the process of relinking the entire application. In large applications, you can save the time and trouble that otherwise would have been spent relinking the application.

The process for using object level PDF is essentially the same as the standard PDF process but with a small change to the `-qpdf2` step. For object level PDF, compile your application using `-qpdf1`, execute the application with representative data, compile the application again with `-qpdf2` but now also use the `-qnoipa` option so that the linking step is skipped.

The steps below outline this process:

1. Compile your application using `-qpdf1`. For example:

   ```bash
   xlf -c -03 -qpdf1 file1.f file2.f file3.f
   ```

   In this example, we are using the option `-O3` to indicate that we want a moderate level of optimization.

2. Link the object files to get an instrumented executable.

   ```bash
   xlf -03 -qpdf1 file1.o file2.o file3.o
   ```

   **Note:** you must use the same optimization options. In this example, the optimization option `-O3`.

3. Run the instrumented executable with sample data that is representative of the data you want to optimize for.

   ```bash
   a.out < sample_data
   ```

4. Compile the application again using `-qpdf2`. Specify the `-qnoipa` option so that the linking step is skipped and PDF optimization is applied to the object files rather than to the entire executable. **Note:** you must use the same optimization options as in the previous steps. In this example, the optimization option `-O3`.

   ```bash
   xlf -c -03 -qpdf2 -qnoipa file1.f file2.f file3.f
   ```

   The resulting output of this step are object files optimized for the sample data processed by the original instrumented executable. In this example, the
optimized object files would be file1.o, file2.o, and file3.o. These can be linked using the system loader ld or by omitting the -c option in the -qpdf2 step.

Notes:
• If you want to specify a file name for the profile that is created, use the pdfname suboption in both the -qpdf1 and -qpdf2 steps. For example:
  xlf -O3 -qpdf1=pdfname=myprofile file1.f file2.f file3.f

Without the pdfname suboption, by default the file name will be _pdf; the location of the file will be the current working directory or whatever directory you have set using the PDFDIR environment variable.
• You must use the same optimization options in each compilation and linking step.
• Because -qnoipa needs to be specified in the -qpdf2 step so that linking of your object files is skipped, you will not be able to use interprocedural analysis (IPA) optimizations and object level PDF at the same time.

**Debugging optimized code**

Debugging optimized programs presents special usability problems. Optimization can change the sequence of operations, add or remove code, change variable data locations, and perform other transformations that make it difficult to associate the generated code with the original source statements.

For example:

**Data location issues**
With an optimized program, it is not always certain where the most current value for a variable is located. For example, a value in memory may not be current if the most current value is being stored in a register. Most debuggers are incapable of following the removal of stores to a variable, and to the debugger it appears as though that variable is never updated, or possibly even set. This contrasts with no optimization where all values are flushed back to memory and debugging can be more effective and usable.

**Instruction scheduling issues**
With an optimized program, the compiler may reorder instructions. That is, instructions may not be executed in the order the programmer would expect based on the sequence of lines in their original source code. Also, the sequence of instructions may not be contiguous. As the user steps through their program with a debugger, it may appear as if they are returning to a previously executed line in their code (interleaving of instructions).

**Consolidating variable values**
Optimizations can result in the removal and consolidation of variables. For example, if a program has two expressions that assign the same value to two different variables, the compiler may substitute a single variable. This can inhibit debug usability because a variable that a programmer is expecting to see is no longer available in the optimized program.

There are a couple of different approaches you can take to improve debug capabilities while also optimizing your program:

**Debug non-optimized code first**
Debug a non-optimized version of your program first, then recompile it
with your desired optimization options. See "Debugging before optimization" for some compiler options that are useful in this approach.

**Use -qoptdebug**

When compiling with -O3 optimization or higher, use the compiler option -qoptdebug to generate a pseudocode file that more accurately maps to how instructions and variable values will operate in an optimized program. With this option, when you load your program into a debugger, you will be debugging the pseudocode for the optimized program. See "Using -qoptdebug to help debug optimized programs" on page 17 for more information.

**Understanding different results in optimized programs**

Here are some reasons why an optimized program might produce different results from one that has not undergone the optimization process:

- 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. The optimization process relies on your application conforming to language standards.

- If a program that works without optimization fails when you optimize, 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 can appear to work properly when compiled without optimization, because of the default assumptions the compiler makes, but can fail when you optimize. Similarly, a program can appear to execute correctly after optimization, but fails at lower optimization levels or when run in a different environment.

- A variation on uninitialized storage. Referring to an automatic-storage variable by its address after the owning function has gone out of scope leads to a reference to a memory location that can be overwritten as other auto variables come into scope as new functions are called.

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.

**Debugging before optimization**

First debug your program, then recompile it with your desired optimization options, and test the optimized program before placing the program into production. If the optimized code does not produce the expected results, you can attempt to isolate the specific optimization problems in a debugging session.

The following list presents options that provide specialized information, which can be helpful during the development of optimized code:

- **-qsmp=noopt**
  
  If you are debugging SMP code, **-qsmp=noopt** ensures that the compiler performs only the minimum transformations necessary to parallelize your code and preserves maximum debug capability.
-qkeepparm
Ensures that procedure parameters are stored on the stack even during optimization. This can negatively impact execution performance. The -qkeepparm option then provides access to the values of incoming parameters to tools, such as debuggers, simply by preserving those values on the stack.

-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. For -qreport to generate a listing, the options -qhot or -qsmp should also be 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.

You can also use the SNAPSHlOT directive to ensure to that certain variables are visible to the debugger at points in your application.

Using -qoptdebug to help debug optimized programs
The purpose of the -qoptdebug compiler option is to aid the debugging of optimized programs. It does this by creating pseudocode that maps more closely to the instructions and values of an optimized program than the original source code. When a program compiled with this option is loaded into a debugger, you will be debugging the pseudocode rather than your original source. By making optimizations explicit in pseudocode, you can gain a better understanding of how your program is really behaving under optimization. Files containing the pseudocode for your program will be generated with the file suffix .optdbg. Only line debugging is supported for this feature.

Compile your program as in the following example:
```bash
xlf myprogram.f -O3 -qhot -g -qoptdebug
```

In this example, your source file will be compiled to a.out. The pseudocode for the optimized program will be written to a file called myprogram.optdbg which can be referred to while debugging your program.

Notes:
- The -g or the -qlinedebug option must also be specified in order for the compiled executable to be debuggable. However, if neither of these options are specified, the pseudocode file <output_file>.optdbg containing the optimized pseudocode will still be generated.
- The -qoptdebug option only has an effect when one or more of the optimization options -qhot, -qsmp, -qipa, or -qpdf are specified, or when the optimization
levels that imply these options are specified; that is, the optimization levels -O3, -O4, and -O5. The example shows the optimization options -qhot and -O3.

**Debugging the optimized program**

See the figures below as an aid to understanding how the compiler may apply optimizations to a simple program and how debugging it would differ from debugging your original source.

**Figure 4** Represents the original non-optimized code for a simple program. It presents a couple of optimization opportunities to the compiler. For example, the variables z and d are both assigned by the equivalent expressions x + y. Therefore, these two variables can be consolidated in the optimized source. Also, the loop can be unrolled. In the optimized source, you would see iterations of the loop listed explicitly.

**Figure 5 on page 19** Represents a listing of the optimized source as shown in the debugger. Note the unrolled loop and the consolidation of values assigned by the x + y expression.

**Figure 6 on page 20** Shows an example of stepping through the optimized source using the debugger. Note, there is no longer a correspondence between the line numbers for these statements in the optimized source as compared to the line numbers in the original source.

```fortran
FUNCTION FOO(X, Y)
    Z = X + Y
    D = X + Y
    DO I = 1, 4
        PRINT *, D, Z
    END DO
    PRINT *, D, Z
END FUNCTION

PROGRAM MAIN
    CALL FOO(3.0, 4.0)
END PROGRAM MAIN
```

*Figure 4. Original code*
REAL*4 FUNCTION foo (x, y)
  @CSE2 = x
  @CSE1 = y
  #2 = _xlfBeginIO(6,257,#1,1024,NULL,0,NULL)
  @CSE0 = @CSE2 + @CSE1
  #3 = @CSE0
  CALL _xlfWriteLDReal(%VAL(#2),#3,4,4)
  #4 = @CSE0
  CALL _xlfWriteLDReal(%VAL(#2),#4,4,4)
  _xlfEndIO(%VAL(#2))
  #2 = _xlfBeginIO(6,257,#1,1024,NULL,0,NULL)
  #3 = @CSE0
  CALL _xlfWriteLDReal(%VAL(#2),#3,4,4)
  #4 = @CSE0
  CALL _xlfWriteLDReal(%VAL(#2),#4,4,4)
  _xlfEndIO(%VAL(#2))
  #2 = _xlfBeginIO(6,257,#1,1024,NULL,0,NULL)
  #3 = @CSE0
  CALL _xlfWriteLDReal(%VAL(#2),#3,4,4)
  #4 = @CSE0
  CALL _xlfWriteLDReal(%VAL(#2),#4,4,4)
  _xlfEndIO(%VAL(#2))
  #2 = _xlfBeginIO(6,257,#1,1024,NULL,0,NULL)
  #3 = @CSE0
  CALL _xlfWriteLDReal(%VAL(#2),#3,4,4)
  #4 = @CSE0
  CALL _xlfWriteLDReal(%VAL(#2),#4,4,4)
  _xlfEndIO(%VAL(#2))
  RETURN
END FUNCTION foo

PROGRAM main ()
  T_3 = 3.00000000E+00
  T_4 = 4.00000000E+00
  CALL foo(T_3,T_4)
  CALL _xlfExit(0)
END PROGRAM main

Figure 5. dbx debugger listing
Getting more performance

The XL compiler family offers other strategies and tuning alternatives for increasing performance.

Whether you are already optimizing at `-O5` or you are looking for more opportunities to increase performance without the resource costs of optimizing at higher levels, the XL compiler family offers other strategies and tuning alternatives. For more information, see the following topics:

- Tuning XL compiler applications
- Advanced optimization concepts
- Optimizing your SMP code

Beyond performance: effective programming techniques

Applications that perform well begin with applications that are written well. See the following topics for information about writing better code; whether your goal is to make your code more portable, more easily optimized, or interoperable with other languages.

- Chapter 5, “Compiler-friendly programming techniques,” on page 43
- Chapter 4, “Managing code size,” on page 37
- Chapter 7, “Parallel programming with XL Fortran,” on page 57
- Chapter 8, “Interlanguage calls,” on page 213
Chapter 2. Tuning XL compiler applications

Included as part of the XL Fortran optimization suite are options you can use to instruct the compiler to generate code that executes optimally on a given processor or architecture family, and to instruct the compiler on the execution characteristics of your application.

The better you can convey those characteristics, the more precisely the compiler can tune and optimize your application. This section assumes that you have already begun optimizing your application using the strategies found in Optimizing your applications.

Tuning for your target architecture

By default, the compiler generates code that runs on all supported systems, though this code does not run optimally on all supported systems. By selecting options to target the appropriate architectures, you can optimize your application to suit the broadest possible selection of relevant processors, a range of processors within a given family, or a specific processor.

The compiler options in the Options for targeting your architecture table introduce how you can control optimizations affecting individual aspects of your target architecture. This section also goes into further detail on how you can use some of those options to ensure your application provides the best possible performance on those targets.

Table 8. Options for targeting your architecture

<table>
<thead>
<tr>
<th>Option</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>-q32</td>
<td>Generates code for a 32-bit addressing model (32-bit execution mode).</td>
</tr>
<tr>
<td>-q64</td>
<td>Generates code for a 64-bit addressing model (64-bit execution mode).</td>
</tr>
<tr>
<td>-qarch</td>
<td>Selects a family of processor architectures, or a specific architecture that the compiler will generate machine instructions for. If you specify multiple architecture settings, only the last architecture is considered valid.</td>
</tr>
<tr>
<td>-qtune</td>
<td>Focuses optimizations for execution on a given processor without restricting the processor architectures that your application can execute on. If you specify multiple architecture settings, only the last architecture is considered valid.</td>
</tr>
<tr>
<td>-qcache</td>
<td>Defines a specific cache or memory geometry. Selecting a predefined optimization level like -O2 sets default values for -qcache suboptions.</td>
</tr>
<tr>
<td>-qipa=clonearch</td>
<td>Instructs the compiler to generate duplicate subprograms with each tuned to a particular architecture.</td>
</tr>
</tbody>
</table>

In addition to targeting the correct architecture for your application, it is important to select the right level of optimization. Combining the appropriate architecture settings with an optimization level that fits your application can vastly enhance performance. If you have not already done so, consult Optimizing your applications in addition to this section.
Using -qarch

Using -qarch you can select a machine architecture or a family of architectures on which you can run your application. Selecting the correct -qarch suboption is crucial to influencing chip-level optimization as the choice of -qarch suboption controls:

- The list of machine instructions available to the compiler when generating object code.
- The characteristics and capabilities of the hardware the compiler will model when optimizing.
- Optimization trade-offs and opportunities in individual instruction selection and instruction sequence selection
- The default setting of the -qtune option.

Architecture selection is important at all optimization levels. Even at low optimization levels like -O0 and -O2, specifying the correct target architecture can be beneficial to performance. Specifying the correct target allows the compiler to select more efficient machine instructions and generate instruction sequences that perform best for a particular machine.

The -qarch suboptions allow you to specify individual processors or a family of processors with common instruction sets or subsets. The choice of processor gives you the flexibility of compiling your application to execute optimally on a particular machine, or to execute on a wide variety of machines while still applying as much architecture-specific optimization as possible. The less specific your choice of architecture, the fewer machine instructions available to the compiler when generating code. A less specific choice can also limit the number of hardware intrinsic functions available to your application. A more specific choice of architecture, can make available more instructions and hardware intrinsic functions. The XL Fortran Compiler Reference details the specific chip architectures and architecture families available.

When compiling your application, using a consistent or compatible -qarch setting for all files will ensure that you are getting the most from your architecture targets. If you are using -qipa link-time optimizations, the architecture setting you specify on the link step overrides the compile step setting.

You must ensure that your application executes only on machines that support your -qarch settings. Executing your application on other machines can produce incorrect results, even if your application appears to run without trapping. In some cases, -qarch suboptions are both individual targets and family targets because the instruction set of newer chips is a superset of the instruction set that earlier chips support. For example, the PWR3 -qarch setting can also safely target PWR3, PWR4, and PWR5, and even PPC970 systems because those processors support the complete base PWR3 instruction set.

Choosing the best -qarch suboption

If your application executes on a single type of processor, use the -qarch setting matching your target processor. If your application will run on multiple processor types, choose a -qarch setting with the largest common intersection of all the processors. You can do this by examining the instruction groups available to the processors and choosing a family setting that best represents it. The following table can assist you with that choice. Note that not all XL compilers support all architectures.
### Table 9. Instruction groups for a supported architecture

<table>
<thead>
<tr>
<th><code>-qarch</code> suboption</th>
<th>PowerPC</th>
<th>Graphics</th>
<th>Sqr</th>
<th>64-bit</th>
<th>PWR3</th>
<th>PWR4</th>
<th>PWR5</th>
<th>VMX</th>
<th>PWR6 Architected</th>
<th>PWR6 Raw</th>
</tr>
</thead>
<tbody>
<tr>
<td>ppc family</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ppcgr family</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>604 chip</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ppc64 family</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rs64a</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ppc64gr</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ppc64grsq</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rs64b</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rs64c</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pwr3 chip and family</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pwr4 chip and family</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pwr5 chip and family</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pwr5x chip</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ppc64v family</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ppc64v and family</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pwr6 chip</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pwr6 family</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>qarch</code> suboption</td>
<td>PowerPC</td>
<td>Graphics</td>
<td>Sqr</td>
<td>64-bit</td>
<td>PWR3</td>
<td>PWR4</td>
<td>PWR5</td>
<td>VMX</td>
<td>PWR6 Architected</td>
<td>PWR6 Raw</td>
</tr>
</tbody>
</table>

#### Unsupported architectures

The *Instruction groups for an unsupported architecture* table lists architectures that the compiler no longer supports. Although the compiler will still recognize and emit code for these architectures, the particular behaviors for these settings that previous versions of the compiler support can differ in some instances. Use with discretion.

### Table 10. Instruction groups for an unsupported architecture

<table>
<thead>
<tr>
<th><code>-qarch</code> suboption</th>
<th>Available</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PWR2</td>
<td>PowerPC</td>
</tr>
<tr>
<td>com family</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pwr chip</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pwr2 chip</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>pwr2s chip</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>pwr2sc chip</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>601 chip</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

Chapter 2. Tuning XL compiler applications 23
Using the default value for -qarch represents the broadest possible range of machines that the compiler supports. For example, in 32-bit mode, the defaults to a setting of ppc. As you can see from the Instruction group support by architecture table, ppc limits the available instructions. If you know that your code will only execute on POWER3™ or newer machines, avoid the default -qarch setting and choose at least PWR3.

If you require optimal performance on multiple differing machines running the same copy of your application, you can use -qipa=clonearch. This option instructs the compiler to generate duplicate subprograms with each tuned to a particular architecture.

Other options and -qarch
Other compiler options can influence the suboption selection for -qarch. The -q64 option forces an upgrade of the -qarch suboption to the minimum chip that can support 64-bit instructions. For example, on AIX, the setting is PPC64. The -qarch=auto suboption is selected automatically when you compile at -O4 and -O5, and assumes that your compilation machine and your target execution machine are the same. For example, if you compile on a POWER5-based system and specify -O5, the -qarch setting defaults to PWR5. You can override this behavior by specifying the -qarch option after the -O4 or -O5 compiler options.

Using -qtune
The -qtune option focuses optimizations for execution on a given processor without restricting the processor architectures that your application can execute on, generating machine instructions consistent with your -qarch architecture choice. Using -qtune also guides the optimizer in performing transformations, such as instruction scheduling, so that the resulting code executes most efficiently on your chosen -qtune architecture. The -qtune option tunes code to run on one particular processor architecture, and includes only specific processors as suboptions. The -qtune option does not support suboptions representing families of processors.

Use -qtune to specify the most common or critical processor where your application executes. For example, if your application usually executes on POWER5™ based systems, but will sometimes execute on a POWER4™ based system, specify -qtune=pwr5. The compiler generates code that executes more efficiently on a POWER5 based system, but will still run correctly on a POWER4 based system.

The default -qtune setting depends on the -qarch setting. If the -qarch option is set to a particular machine architecture, this limits the range of available -qtune suboptions, and the default tune setting will be compatible with the selected target processor. If -qarch option is set to a family of processors, the range of values available for -qtune expands across that family, and the default is chosen from a commonly used machine in that family. If you compile with -qtune=auto, the default for optimization levels -O4 and -O5, the compiler detects the machine characteristics on which you are compiling, and assumes you want to tune for that type of machine. You can override this behavior by specifying -qtune after the -O4 or -O5 compiler options.
If you need to create a single binary that will run on a range of PowerPC hardware, consider using the `-qtune=balanced` option. With this option in effect, optimization decisions made by the compiler are not targeted to a specific version of hardware. Instead, tuning decisions try to include features that are generally helpful across a broad range of hardware and avoid those optimizations that may be harmful on some hardware. Note that you should verify the performance of code compiled with the `-qtune=balanced` option before distributing it.

**Using -qcach**
The `-qcach` option allows you to instruct the optimizer on the memory cache layout of your target architecture. There are several suboptions you can specify to describe cache characteristics such as:

- The types of cache available
- The cache size
- Cache-miss penalties

The `-qcach` option is only effective if you understand the cache characteristics of the execution environment of your application. Before using `-qcach`, look at the options section of the listing file with the `-qlist` option to see if the current cache settings are acceptable. The settings appear in the listing when you compile with `-qlistopt`. If you are unsure about how to interpret this information, do not use `-qcach`, and allow the compiler to use default cache settings.

If you do not specify `-qcach`, the compiler makes cache assumptions based on your `-qarch` and `-qtune` settings. If you compile with the `-qcach=auto` suboption, the default at optimization levels `-O4` and `-O5`, the compiler detects the cache characteristics of your compilation machine and tunes cache optimizations for that cache layout. If you do specify `-qcach`, also specify `-qhot`, or an option such as `-O4` that implies `-qhot`. The optimizations that `-qhot` performs are designed to take advantage of your `-qcach` settings.

**Before you finish tuning**
Consult the following list to ensure that you are getting the most out of your target machine options.

- Do not specify a `-qarch` option that is incompatible with your hardware. This can produce unexpected results.
- Specify a `-qarch` setting that represents the largest common instruction set available to the machines that your application will execute on. Consult the `Instruction group support by architecture` table for more information.
- If you are executing your application on multiple machines, choose the `-qtune` suboption that aligns with the machine you expect your application to run on most frequently or where performance is most important.
- If compiling with `-qcach`, specify `-qhot` as well, which can take advantage of your cache settings.

**Further option driven tuning**
You can use options to convey the characteristics of your application to the compiler, tuning the optimizations that the compiler will apply. Option driven tuning is a process that can require experimentation to find the right combination of options to increase the performance of your application.

The XL compilers support many options that allow you to assert that your application will not follow certain standard language rules in some instances. The
compiler assumes language standard compliance and can perform unsafe
optimizations if your application is not compliant. Standards-conforming
applications are more easily optimized and more portable, but when full
compliance is not possible, use the appropriate options to ensure your code is
optimized safely.

For complete compiler option syntax, see the XL Fortran Compiler Reference.

Options for providing application characteristics

This section provides a list of options that can dictate a wide variety of
characteristics about your application to the compiler including floating-point and
loop behaviors.

Option Description

-qalias
Supports several suboptions that can help the compiler analyze the
characteristics of your application. For more information on aliasing, see
Advanced optimization concepts.

noaryovrlp
Asserts that your application contains no array assignments
between storage associated (overlapping) arrays.

nointptr
Asserts that your application does not make use of integer (Cray)
pointers.

nopteovrlp
Asserts that your application does not contain pointee variables
that refer to any data objects that are not pointee variables. Also,
that your application does not contain two pointee variables that
can refer to the same storage location.

std
Asserts that your application follows all language rules for variable
aliasing. This is the default compiler setting. Specify -qalias=nostd
if your application does not follow all variable aliasing rules.

-qassert
Includes the following suboptions that can be useful for providing some
loop characteristics of your application.

nodeps
Asserts that the loops in your application do not contain loop carry
dependencies.

itercnt={number}
Gives the optimizer a value to use when estimating the number of
iterations for loops where it cannot determine that value.

-qddim
Forces the compiler to reevaluate the bounds of a pointee array each time
the application references the array. Specify this option only if your
application performs dynamic dimensioning of pointee arrays.

-qdirectstorage
Asserts that your application accesses write-through-enabled or
cache-inhibited storage.
-qfloat

Provides the compiler with floating-point characteristics for your application. The following suboptions are particularly useful.

- **nans** 
  Asserts that your application makes use of signaling NaN (not-a-number) floating-point values. Normal floating-point operations do not create these values, your application must create signalling NaNs.

- **rrm** 
  Prohibits optimization transformations that assume the floating-point rounding mode must be the default setting round-to-nearest. If your application changes the rounding mode in any way, specify this option.

-qflttrap

Offers you the ability to control various aspects of floating-point exception handling that your application can require if it attempts to detect or handle such exceptions.

-qieee

Specifies the preferred floating-point rounding mode when evaluating expressions at compile time. This option is important if your application requires a non-default rounding mode in order to have consistency between compile-time evaluation and runtime evaluation.

You can also specify -y to set the preferred floating-point rounding mode.

-qlargepage

Indicates that your application is designed to execute in the AIX 16 MB large page memory environment.

-qlibansi

Asserts that any external function calls in your compilation that have the same name as standard C library function calls, such as malloc or memcpy, are in fact those functions and are not a user-written function with that name.

-qlibbssl

Asserts that your application will be linked with IBM's ESSL high-performance mathematical library and that mathematical operations can be transformed into calls to that library. For more information on ESSL, see the [High performance libraries](#) topic.

-qlibposix

Asserts that any external function calls in your application that have the same name as standard Posix library function calls are in fact those functions and are not a user-written function with that name.

-qonetrip

Asserts that all DO loops in your application will execute at least one iteration. You can also specify this behavior with -1.

-qnostrictieemod

Allows the compiler to relax certain rules required by the Fortran 2003 standard related to the use of the IEEE intrinsic modules. Specify this option if your application does not use these modules.

-qstrict_induction

Prevents optimization transformations that would be unsafe if DO loop integer iteration count variables overflow and become negative. Few applications contain algorithms that require this option.
-qthreaded
Informs the compiler that your application will execute in a multithreaded/SMP environment. Using an _r invocation, like xlf_r, adds this option automatically.

-qnounwind
Informs the compiler that the stack will not be unwound while any routine in your application is active. The -qnounwind option enables prologue tailoring optimization, which reduces the number of saves and restores of nonvolatile registers.

-qnozerosize
Asserts that this application does not require checking for zero-sized arrays when performing array operations.

Options to control optimization transformations
There are many options available to you in addition to the base set found in the Optimizing your applications section. Some of these options prevent an optimization that can be unsafe for certain applications or enable one that is safe for your application, but is not normally available as part of the optimization process.

Option Description

-qcompact
Chooses a reduction of final code size over a reduction in execution time. You can use this option to constrain the optimizations of -O3 and higher. For more information on restriction code size, see the Managing code size section.

-qfdpr Prepares your object code for additional optimization by the FDPR object code optimizer.

-qenablevmx
Allows you to take advantage of the VMX capabilities of chips such as the PPC970.

-qfloat
This option provides a number of suboptions for controlling the optimizations to your floating-point calculations.

-norsqrt Prevents the replacement of the division of the result of a square-root calculation with a multiplication by the reciprocal of the square root.

-nostrictmaf Prevents certain floating-point multiply-and-add instructions where the sign of signed zero value would not be preserved.

-qipa Includes many suboptions that can assist the IPA optimizations while analyzing your application. If you are using the -qipa option or higher optimization levels that imply IPA, it is to your benefit to examine the suboptions available.

-qmaxmem Limits the memory available to certain memory-intensive optimizations at low levels. Specify -qmaxmem=-1 to remove these memory limits.
Options

-qnoprefetch
Prevents the the insertion of prefetching machine instructions into your application during optimization.

-Q
Allows you to exert control over inlining optimization transformations. For more information on inlining, see the Advanced optimization concepts section.

-qsmallstack
Instructs the compiler to limit the use of stack storage in your application. This can increase heap usage.

-qsmpl Produces code for an SMP system. This option also searches for opportunities to increase performance by automatically parallelizing your code. The Parallel programming with XL Fortran section contains more information on writing parallel code.

-qstacktemp
Allows you to limit certain compiler temporaries allocated on the stack. Those not allocated on the stack will be allocated on the heap. This option is useful for applications that use enough stack space to exceed stack user or system limits.

-qstrict
Limits optimizations to strict adherence to implied program semantics. This often prevents the compiler from ignoring certain little-used rules in the IEEE floating-point specification that few applications require for correct behavior. For example, reordering or reassociating a sequence of floating-point calculations can cause floating-point exceptions at an unexpected location or mask them completely. The -qstrict option includes suboptions that refine the control of the transformations performed by the optimizers. Do not use this option unless your application requires strict adherence as -qstrict and its suboptions can severely inhibit optimization.

-qunroll
Allows you to independently control loop unrolling. At -O3 and higher, -qunroll is a default setting.

Options to assist with performance analysis

The compiler provides a set of options that can help you analyze the performance aspects of your application. These options are most useful when you are selecting your level of optimization and tuning the optimization process to the particular characteristics of your application.

-d
Informs the compiler that you want to preserve the preprocessed versions of your compilation files. Typically these files would have a .F extension.

-g
Inserts full debugging information into your object code. While the optimization process can obscure original program meaning, at least some of the information that this option produces is useful to performance analysis tools. You can also specify this behavior with -qdbg.

-p
Inserts appropriate profiling information into your object to code to make using tools for performance analysis possible. You can also specify this behavior with -pg.

-qdpcl Prepares your object for processing by tools based on the Dynamic Probe Class Library (DPCL).
-qlinedebug
An option similar to -g, this option inserts only minimal debug information into your object code such as function names and line number information.

-qlist
Produces a listing file containing a pseudo-assembly listing of your object code.

-qreport
Inserts information in the listing file showing the transformations done by certain optimizations.

-S
Produces a .s file containing the assembly version of the .o file produced by the compilation.

-qshowpdf
If you specify this option with -qpdf1 and a minimum of -O optimization, the optimization process inserts additional information into your application that the showpdf utility can make use of when analyzing the result of a PDF run. For more information on profile directed feedback, see Profile-directed feedback.

-qtbttable
Limits the amount of debugging traceback information in object files, which reduces the size of the program. Use -qtbttable=full if you intend to analyze your application with the tprof profiling utility.

Options that can inhibit performance
Some compiler options are necessary for some applications to produce correct or repeatable results. Usually, these options instruct the compiler to enforce very strict language semantics that few applications require. Others are supported by the compiler to allow compilation of code that does not conform to language standards. Avoid these options if you are trying to increase the runtime performance of your application. In cases where these options are enabled by default, you must disable them to increase performance. You can specify -qlistopt to show, in the listing file, the settings of each of these options.

The following table summarizes the options that can inhibit performance:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-qalias=nostd</td>
<td>Instructs the compiler to enforce very strict language semantics.</td>
</tr>
<tr>
<td>-qfloat=nosqrt</td>
<td>Same as -qalias=nostd.</td>
</tr>
<tr>
<td>-qstacktemp=value</td>
<td>Instructs the compiler to use a specific stack size.</td>
</tr>
<tr>
<td>-qunwind</td>
<td>Instructs the compiler to optimize unwind code.</td>
</tr>
<tr>
<td>-qcompact</td>
<td>Instructs the compiler to optimize compact code.</td>
</tr>
<tr>
<td>-qnoenablevmx</td>
<td>Same as -qcompact.</td>
</tr>
<tr>
<td>-qfloat=nostRICTMAF</td>
<td>Instructs the compiler to optimize float code.</td>
</tr>
<tr>
<td>-qoprefetch</td>
<td>Instructs the compiler to optimize prefetch code.</td>
</tr>
<tr>
<td>-Q!</td>
<td>Instructs the compiler to optimize induction code.</td>
</tr>
<tr>
<td>-qsmallstack</td>
<td>Instructs the compiler to optimize small stack code.</td>
</tr>
<tr>
<td>-qnounroll</td>
<td>Instructs the compiler to optimize no unroll code.</td>
</tr>
</tbody>
</table>

Related reference
- See -qalias in the Compiler Reference
- See -qcompact in the Compiler Reference
- See -qnoenablevmx in the Compiler Reference
- See -qfloat in the Compiler Reference
- See -qoprefetch in the Compiler Reference
See -Q! in the Compiler Reference
See -qsmallstack in the Compiler Reference
See -qstacktemp in the Compiler Reference
See -qstrict in the Compiler Reference
See -qstrict_induction in the Compiler Reference
See -qstrictieemod in the Compiler Reference
See -qnounroll in the Compiler Reference
See -qunwind in the Compiler Reference
See -qzerosize in the Compiler Reference
Chapter 3. Advanced optimization concepts

After you apply command-line optimizations and tuning that are appropriate to your application and the constraints of your development cycle, you have opportunities to further improve the performance of your application through aliasing and inlining.

Aliasing

An alias occurs when different variables point directly or indirectly to a single area of storage. Aliasing refers to assumptions made during optimization about which variables can point to or occupy the same storage area.

When an alias exists, or the potential for an alias occurs during the optimization process, pessimistic aliasing occurs. This can inhibit optimizations like dead store elimination and loop transformations on aliased variables. Also, pessimistic aliasing can generate additional loads and stores as the compiler must ensure that any changes to the variable that occur through the alias are not lost.

When aliasing occurs there is less opportunity for optimization transformations to occur on and around aliased variables than variables where no aliasing has taken place. For example, if variables A, B, and C are all aliased, any optimization must assume that a store into or a use of A is also a store or a use of B and C, even if that is not the case. Some of the highest optimization levels can improve alias analysis and remove some pessimistic aliases. However, in all cases, when it is not proven during an optimization transformation that an alias can be removed that alias must be left in place.

Where possible, avoid programming techniques that lead to pessimistic aliasing assumptions. These aliasing assumptions are the single most limiting factor to optimization transformations. The following situations can lead to pessimistic aliasing:

- When you assign a pointer the address of any variable, the pointer can be aliased with globally visible variables and with static variables visible in the pointer’s scope.
- When you call a procedure that has dummy arguments passed by reference, aliasing occurs for variables used as actual arguments, and for global variables.
- The compiler will make several worst-case aliasing assumptions concerning variables in common blocks and modules. These assumptions can inhibit optimization.

Some compiler options like -qalias can affect aliasing directly. For more information on how to tune the aliasing behavior in your application, see "Options for providing application characteristics” on page 26.

Inlining

Inlining is the process of replacing a subroutine or function call at the call site with the body of the subroutine or function being called. This eliminates call-linkage overhead and can expose significant optimization opportunities.
For example, with inlining, the optimizer can replace the subroutine parameters in the function body with the actual arguments passed. Inlining trade-offs can include code bloat and an increase in the difficulty of debugging your source code.

If your application contains many calls to small procedures, the procedure call overhead can sometimes increase the execution time of the application considerably. Specifying the -qipa=inline compiler option can reduce this overhead. Additionally, you can use the -p or -pg options and profiling tools to determine which subprograms your application calls most frequently, and list their names using -qipa=inline to ensure inlining.

The -qipa option can perform inlining where the calling and called procedures are in different compilation units.

```
xlf95 -O3 -qipa=inline inline.f
```

```
xlf95 -O3 -qipa=inline called_100_times,called_1000_times inline.f
```

**Finding the right level of inlining**

A common occurrence in application optimization is excessive inlining. This can actually lead to a decrease in performance because running larger programs can cause more frequent cache misses and page faults. Since the XL compilers contain safeguards to prevent excessive inlining, this can lead to situations where subprograms you want to inline are not automatically inlined when you specify -qipa=inline.

Some common conditions that prevent -qipa=inline from inlining particular subprograms are:

- The calling and called procedures are in different compilation units. If so, you can use the -qipa option on the link step to enable cross-file inlining.
- After inlining expands a subprogram to a particular limit, the optimizer does not inline subsequent calls from that subprogram. The limits depend on if the subprogram called is named by a -qipa=inline option.

Consider an example with three procedures where: 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 you would compile as follows:

```
xlf -qipa=inline=c file.f
```

Specifying -qipa=inline means that calls to C are more likely to be inlined. If B and C were twice as large as the upper size limit for automatic inlining, no inlining would take place for calls to B. However inlining would still take place for some calls to C.

- Any interface errors, such as different numbers, sizes, or types of arguments or return values, can prevent inlining for a subprogram call. On AIX you can also compile with the -qextchk option to locate these errors. You can also use interface blocks for the programs being called.
- Actual or potential aliasing of dummy arguments or automatic variables can limit inlining. Consider the following cases:
  - You compile a file containing either the calling or called procedure with -qalias=nostd, and the function takes parameters.
  - There are more than approximately 31 arguments to the procedure your application is calling.
- Any automatic variables in the called procedures are involved in an
  EQUVALENCE statement
- 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.

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 to inline C/C++ functions into Fortran programs and Fortran
functions into C/C++ programs during link time optimizations. You must compile
the C/C++ code using the IBM XL C/C++ compilers with -qipa and a compatible
option set to that used in the IBM XL Fortran compilation.
Chapter 4. Managing code size

Code size is often not a detriment to performance for most XL compiler programmers. For some however, generating compact object code can be as important as generating efficient code.

Oversized programs can affect overall performance by creating a conflict for real storage between pages of virtual storage containing code, and pages of virtual storage containing data. On systems with a small, combined instruction and data cache, cache collisions between code and data can also reduce performance. This section provides suggestions on how to achieve a balance between code efficiency and object-module size, while identifying compiler options that can affect object-module size. Code size tuning is most effective once you have built a stable application and run optimization at-O2 or higher.

Reasons for tuning for code size include:

- Your application design calls for an implementation with limited real memory, instruction-cache space, or disk space.
- When loading your application, it uses enough memory to create a conflict between code areas and data areas in real memory, and both code and data are frequently paged in and out.
- There are high activity areas in your code large enough that instruction cache and instruction Translation Lookaside Buffer (TLB) misses have a major effect on performance.
- You intend your application to run on a host that serves end users, or in a batch environment with limits on real memory.

Before tuning for code size, it is important for you to determine whether code size is the actual problem. Very large applications tend to have small clusters of high activity and large sections of infrequently accessed code. If a particular code page is not accessed in a particular run, that page is never loaded into memory, and has no negative impact on performance. If you are tuning for code size due to the high activity code segments that cause instruction cache and instruction TLB misses that have a major effect on performance, this can be symptomatic of a program structure that requires improvement or hardware not suited to the resource requirements of the application.

If your data takes up more real storage than is available, reducing code size can improve performance by ensuring that fewer pages of data are paged out as code is paged in. However, data blocking strategies are likely to prove both more effective and easier to implement. Processing data in each page as completely as possible before moving on to the next page can reduce the number of data page misses.

If you are coding an application for a machine with a combined instruction and data cache, you can improve performance by applying the techniques described later in this section, but tuning for data cache management can yield better results than code-size tuning. Also note that highly tuning your code for the cache characteristics of one system can lead to undesirable performance results if you execute your application elsewhere.
Steps for reducing code size

Reducing the code size of your application can have a positive affect on the performance of your application.

Consider the following steps for reducing code size:

- Ensure that you have built a stable application that compiles at `-O2` or higher.
- Use performance analysis tools to isolate high activity code segments and tune for performance where appropriate. Basing decisions for code size tuning on an application that has already undergone performance analysis will give you more information on where your application could benefit from code size tuning.
- Use compiler options like `-qipa` that can help reduce code size. See Compiler option influences on code size for more information.
- On AIX, use the `rmsss` command to mimic the memory conditions of your target system. This command can reduce effective real memory size and help you obtain more realistic profiles of your program and identify areas where code size could be a problem on smaller systems.

Be aware that optimization can cause code to expand significantly through loop unrolling, invariant `IF` floating, inlining, and other optimizations. The higher your optimization level, the more code size can increase. For more information on finding an optimization level appropriate for your application, see Chapter 1, “Optimizing your applications,” on page 1.

Compiler option influences on code size

As already noted, high optimization levels can increase code size. You can use other compiler options to influence the size of your code and improve performance.

The `-qipa` compiler option

The `-qipa` option enables interprocedural analysis (IPA) by the compiler. Interprocedural analysis analyzes the relationships between procedures and the code that references those procedures, so that more optimizations within procedures and across procedure references can take place. Interprocedural analysis can decrease code size and improve performance at the same time. In some cases however, IPA inlining can increase code size. Use with discretion.

Related reference

See interprocedural analysis (IPA) in the Compiler Reference

The `-Q` inlining option

Using the `-Q` compiler option, you can specify that the optimizer consider all Fortran 90 or Fortran 95 procedures, or a particular list of procedures for inlining. Specifying `-qipa=inline` also inlines procedures and can alter the limits of `-Q`.

Inlining procedures can increase the performance of your application, though if your program references a procedure from many different locations in the source code, inlining that procedure can increase code size dramatically. You can disable procedure inlining entirely using `-Q!`, or `-qipa=inline=noauto`. You can also partially disable inlining with `-Q-names`.

However, do not assume that all inlining increases code size. When your source code references a very small procedure a large number of times, inlining can reduce code size, as inlining eliminates control transfer and data interface code. In
addition, inlining code facilitates other optimizations at the point of inlining, by providing information on the values of arguments referencing the procedure. If a procedure is very small and is referenced from a number of places, inlining can also increase code locality and reduce code paging.

The **-qhot** compiler option

The loop analysis and optimization available when you specify **-qhot** can increase code size. If your application contains many large loops and loop optimization opportunities exist, **-qhot** code size can increase significantly along with performance. Specifying **-qhot=level=0** will perform minimal high-order transformations if code size is an issue. The section on [High-order transformation](#) contains more information on using **-qhot** effectively.

The **-qcompact** compiler option

The **-qcompact** compiler option instructs the compiler to avoid certain optimizing transformations that expand the object code. Compiling with **-qcompact**, disables many transformations, including:

- Loop unrolling
- Expansion of fixed-point multiply by more than one instruction
- Inline expansion of some string and memory manipulation functions. In some cases **-qcompact** will avoid inlining opportunities entirely.

Specifying **-qcompact** creates a trade-off between the performance of individual routines in your application, and overall system performance. Suppressing transformations degrades the performance of individual routines, while overall system performance can increase as a more compact program can provide some or all of the following:

- Fewer instruction-cache misses
- Fewer TLB misses for pages of application code
- Fewer page faults for application code

**Other influences on code size**

In addition to compiler options, there are a number of ways programming and analysis can influence the size of your source code.

**High activity areas**

Once you apply the techniques discussed earlier in this section, your strategy for further code size reduction depends on your objective. Use profiling tools to locate hot spots in your program; then follow one of the following guidelines:

- If you want to reduce code size to reduce program paging, concentrate on minimizing branches and procedure references within those hot spots.
- If you want to reduce code size to reduce the size of your program’s files on disk, concentrate on areas that are not hot spots. Remove any expansive optimizations from code that does not contain hot spots.

**Computed GOTOs and CASE constructs**

A sparse computed GOTO can increase code size considerably. In a sparse computed GOTO, most statement labels point to the default. Consider the following example where label 10 is the default:
Although fewer cases are shown, the following **CASE** construct is a functionally equivalent to the example above. \( N \) is the value of the largest integer that the computed **GOTO** or **CASE** construct is testing.

```fortran
INTEGER IA(10000)
SELECT CASE (IA(I))
  CASE DEFAULT
      GOTO 10
  CASE (5)
      GOTO 20
  CASE (10)
      GOTO 30
  CASE (11)
      GOTO 20
  ! ...
  CASE (N-10)
      GOTO 30
  CASE (N-2)
      GOTO 20
  CASE (N)
      GOTO 30
END SELECT
```

In both examples, the compiler builds a branch table in the object file that contains one entry for each possibility from 1 to \( N \), where \( N \) is the largest integer value tested. The data section of the program stores this branch table. If \( N \) is very large, the table can increase both the size of the object file and the effects of data-cache misses.

If you use a **CASE** construct with a small number of cases and wide gaps between the test values of the cases, the compiler selects a different algorithm to dispatch to the appropriate location, and the resulting code can be more compact than a functionally equivalent computed **GOTO**. The compiler cannot determine that a computed **GOTO** has a default branch point, so the compiler assumes that any value in the range will be selected. In a **CASE** construct, the compiler assumes that cases you do not specify in the construct are handled as default.

**Code size with dynamic or static linking**

Dynamic or static linking each affect the size of your code, and the resulting performance of your application.

**Dynamic linking and code size**

When linking your XL compiler programs, dynamic linking often ensures more compact code than linking statically. Dynamic linking does not include library procedures in your object file. Instead, a reference at runtime causes the operating system to locate the dynamic library that contains the procedure, and reference that procedure from the library of the operating system. Only one copy of the procedure is in memory, even if several programs, or copies of a single program,
are accessing the procedure simultaneously. This can reduce paging overhead. However, any libraries your program references must be present in your application’s execution environment, or ship with your application.

Note that if your program references high performance libraries like BLAS or ESSL, these procedures are dynamically linked to your program by default.

**Static linking and code size**

Static linking binds library procedures into your application’s object file. This can increase the size of your object file. If your program references only a small portion of the procedures available in a library, static linking can eliminate the need to provide the library to your users. However, static linking ties your application to one version of the library which can be detrimental in situations where your application will execute in different environments, such as different levels of the operating system.
Chapter 5. Compiler-friendly programming techniques

Writing compiler-friendly code, with both the optimizer and portability in mind, can be as important to the performance of your application as the compilation options that you specify.

General practices

It is not necessary to hand-optimize your code, as hand-optimizing can introduce unusual constructs that can obscure the intentions of your application from the compiler and limit optimization opportunities.

Large programs, especially those that take advantage of 64-bit capabilities, can use significant address space resources. Use 64-bit mode only if your application requires the additional address space resources it provides you with.

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, compiling with -qipa can help minimize the impact on performance. Attempting to optimize an application with many small functions without the benefit of -qipa can severely limit the scope of other optimizations.

Use command invocations like xlf90 and xlf95, which imply -qnosave, and -qnosave sets the default storage class of all variables to automatic. This provides more opportunities for optimization. All compiler command invocations except f77, fort77, xlf, xlf_r, and xlf_r7 use -qnosave by default.

Use modules to group related subroutines and functions.

Use module variables instead of common blocks for global storage.

Mark all code that accesses or manipulates data objects by independent I/O processes and independent, asynchronously interrupting processes as VOLATILE. For example, mark code that accesses shared variables and pointers to shared variables. Mark your code carefully however, as VOLATILE is a barrier to optimization as accessing a VOLATILE object forces the compiler to always load the value from storage. This prevents powerful optimizations such as constant propagation or invariant code motion.

The XL compilers support [high performance libraries](#) that can provide significant advantages over custom implementations or generic libraries.

Variables and pointers

The effective use of aliasing and of variables and pointers provides opportunities for improved performance and further optimization.

Obey all aliasing rules. Avoid specifying -qalias=nostd. For more information on aliasing and how it can affect performance, see "Aliasing" on page 33.

Avoid unnecessary use of global variables and pointers, including module variables and common blocks. When using global variables and pointers in a loop,
load them into a local variable before the loop and store them back after. If you do not use the local variable somewhere other than in the loop body, the optimization process can usually recognize what you are doing and expose more optimization opportunities. Replacing a global variable in a loop with a local variable reduces the possibilities for aliasing.

Use the INTENT statement to describe the usage of dummy arguments.

Limit the use of ALLOCATABLE objects and POINTER variables to situations demanding dynamic memory allocation.

**Arrays**

Where possible, use local variables instead of global variables for loop index variables and bounds.

Whenever possible, ensure references to arrays or array sections refer to contiguous blocks of storage. Noncontiguous memory array references, when passed as parameters, lead to copy-in and copy-out operations.

Keep your array expressions simple so that the optimizer can deduce access patterns more easily and reuse index calculations in whole or in part.

Frequent use of array-to-array assignment and WHERE constructs can impact performance by increasing temporary storage and creating loops. Using -qlist or -qreport can help you understand the performance characteristics of your code, and where applying -qhot could be beneficial. If you are already optimizing with -qipa, ensure you are using the list=filename option, so that the -qlist listing file is not overwritten.

**Choosing appropriate variable sizes**

Improve the efficiency of your application by choosing the appropriate variable sizes.

When programming SMP applications, use the CONTAINS statement only to share thread local storage.

In most cases using INTEGER(4) in 32-bit mode and INTEGER(8) in 64-bit mode for scalars improves the efficiency of mathematical calculations and calling conventions when passing objects. However, if your code contains large arrays with values that can fit in an INTEGER(1) or INTEGER(2) in 32-bit mode, or an INTEGER(4) in 64-bit mode, using smaller kind parameters can actually improve memory efficiency by reducing memory traffic to load or store data.

Use the lowest floating-point precision appropriate to your application. Higher precisions can reduce performance, so use the REAL(16), or COMPLEX(16) data types only when you require extremely high precision.

On systems with VMX, using REAL(4) and -qhot=simd provides opportunities for short vectorization not available with larger floating-point types.
Chapter 6. High performance libraries

XL Fortran is shipped with a set of libraries for high-performance mathematical computing.

The set of libraries for high-performance mathematical computing are:

- The Mathematical Acceleration Subsystem (MASS) is a set of libraries of tuned mathematical intrinsic routines that provide improved performance over the corresponding standard system math library routines. MASS is described in "Using the Mathematical Acceleration Subsystem libraries (MASS)."

- The Basic Linear Algebra Subprograms (BLAS) are a subset of routines from IBM’s Engineering and Scientific Subroutine Library (ESSL) library, which provides matrix/vector multiplication functions tuned for PowerPC architectures. The BLAS functions are described in "Using the Basic Linear Algebra Subprograms – BLAS\(^{\text{TM}}\) on page 53.

Note that if you are going to link your application with the ESSL libraries, using -qessl and IPA allows the optimizer to automatically use ESSL routines.

Using the Mathematical Acceleration Subsystem libraries (MASS)

The MASS libraries consist of a library of scalar XL Fortran routines described in "Using the scalar library" and a set of vector libraries tuned for specific architectures, described in "Using the vector libraries" on page 47. The functions contained in both scalar and vector libraries are automatically called at certain levels of optimization, but you can also call them explicitly in your programs. Note that the accuracy and exception handling might not be identical in MASS functions and system library functions.

"Compiling and linking a program with MASS” on page 52 describes how to compile and link a program that uses the MASS libraries, and how to selectively use the MASS scalar library functions in conjunction with the regular system libraries.

Related external information


Using the scalar library

The MASS scalar library libmass.a contains an accelerated set of frequently used math intrinsic functions that provide improved performance over the corresponding standard system library functions. The MASS scalar functions are used when explicitly linking libmass.a, but are also available automatically when you compile programs with any of the following options:

- -qhot -qnostrict
- -qhot -O3
- -O4
- -O5

With these options, the compiler automatically uses the faster MASS functions for most math library functions. In fact, the compiler first tries to "vectorize" calls to
math library functions by replacing them with the equivalent MASS vector functions; if it cannot do so, it uses the MASS scalar functions. When the compiler performs this automatic replacement of math library functions, it uses versions of the MASS functions contained in the system library libxlopt.a. You do not need to add any special calls to the MASS functions in your code, or to link to the libxlopt library.

If you are not using any of the optimization options listed above, and want to explicitly call the MASS scalar functions, you can do so as follows:

1. Link the MASS scalar library libmass.a with your application. For instructions, see "Compiling and linking a program with MASS" on page 52
2. All the MASS scalar routines, except those listed in step 3 are recognized by XL Fortran as intrinsic functions, so no explicit interface block is needed. To provide an interface block for the functions listed in step 3 include mass.include in your source file.
3. Include mass.include in your source file for the following functions:
   - acosf, acoshf, asinf, asinhf, atan2f, atanf, atanhf, atanhf, cbrtf, cbrtf, copyf, copysignf, cosf, coshf, cosinf, erff, erfcf, expf, expm1f, hypotf, log10f, log10f, log1pf, rsqrtf, sinf, sincos, sinhf, tanf, tanhf, and x**y

The MASS scalar functions accept double-precision parameters and return a double-precision result, or accept single-precision parameters and return a single-precision result, except sincos which gives 2 double-precision results. They are summarized in Table 12.

Table 12. MASS scalar functions

<table>
<thead>
<tr>
<th>Double-precision function</th>
<th>Single-precision function</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>acos</td>
<td>acosf</td>
<td>(x)</td>
<td>Returns the arccosine of x</td>
</tr>
<tr>
<td>acosh</td>
<td>acoshf</td>
<td>(x)</td>
<td>Returns the hyperbolic arccosine of x</td>
</tr>
<tr>
<td>anint</td>
<td></td>
<td>(x)</td>
<td>Returns the rounded integer value of x</td>
</tr>
<tr>
<td>asin</td>
<td>asinf</td>
<td>(x)</td>
<td>Returns the arcsine of x</td>
</tr>
<tr>
<td>asinh</td>
<td>asinhf</td>
<td>(x)</td>
<td>Returns the hyperbolic arcsine of x</td>
</tr>
<tr>
<td>atan2</td>
<td>atan2f</td>
<td>(x,y)</td>
<td>Returns the arctangent of x/y</td>
</tr>
<tr>
<td>atan</td>
<td>atanf</td>
<td>(x)</td>
<td>Returns the arctangent of x</td>
</tr>
<tr>
<td>atanh</td>
<td>atanhf</td>
<td>(x)</td>
<td>Returns the hyperbolic arctangent of x</td>
</tr>
<tr>
<td>cbrtf</td>
<td></td>
<td>(x)</td>
<td>Returns the cube root of x</td>
</tr>
<tr>
<td>copysignf</td>
<td></td>
<td>(x,y)</td>
<td>Returns x with the sign of y</td>
</tr>
<tr>
<td>cos</td>
<td>cosf</td>
<td>(x)</td>
<td>Returns the cosine of x</td>
</tr>
<tr>
<td>coshf</td>
<td></td>
<td>(x)</td>
<td>Returns the hyperbolic cosine of x</td>
</tr>
<tr>
<td>cosin</td>
<td></td>
<td>(x)</td>
<td>Returns a complex number with the real part the cosine of x and the imaginary part the sine of x.</td>
</tr>
<tr>
<td>drint</td>
<td></td>
<td>(x)</td>
<td>Returns the nearest integer to x (as a double)</td>
</tr>
<tr>
<td>erf</td>
<td>erff</td>
<td>(x)</td>
<td>Returns the error function of x</td>
</tr>
<tr>
<td>erfc</td>
<td>erfcf</td>
<td>(x)</td>
<td>Returns the complementary error function of x</td>
</tr>
</tbody>
</table>
Table 12. MASS scalar functions (continued)

<table>
<thead>
<tr>
<th>Double-precision function</th>
<th>Single-precision function</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exp</td>
<td>expf</td>
<td>(x)</td>
<td>Returns the exponential function of x</td>
</tr>
<tr>
<td>expm1</td>
<td>expm1f</td>
<td>(x)</td>
<td>Returns (the exponential function of x) - 1</td>
</tr>
<tr>
<td>hypot</td>
<td>hypotf</td>
<td>(x,y)</td>
<td>Returns the square root of x^2 + y^2</td>
</tr>
<tr>
<td>lgamma</td>
<td>lgammaf</td>
<td>(x)</td>
<td>Returns the natural logarithm of the absolute value of the Gamma function of x</td>
</tr>
<tr>
<td>log</td>
<td>logf</td>
<td>(x)</td>
<td>Returns the natural logarithm of x</td>
</tr>
<tr>
<td>log10</td>
<td>log10f</td>
<td>(x)</td>
<td>Returns the base 10 logarithm of x</td>
</tr>
<tr>
<td>log1p</td>
<td>log1pf</td>
<td>(x)</td>
<td>Returns the natural logarithm of (x + 1)</td>
</tr>
<tr>
<td>rsqrt</td>
<td></td>
<td>(x)</td>
<td>Returns the reciprocal of the square root of x</td>
</tr>
<tr>
<td>sin</td>
<td>sinf</td>
<td>(x)</td>
<td>Returns the sine of x</td>
</tr>
<tr>
<td>sincos</td>
<td></td>
<td>(x,s,c)</td>
<td>Sets s to the sine of x and c to the cosine of x</td>
</tr>
<tr>
<td>sinh</td>
<td>sinhf</td>
<td>(x)</td>
<td>Returns the hyperbolic sine of x</td>
</tr>
<tr>
<td>sqrt</td>
<td></td>
<td>(x)</td>
<td>Returns the square root of x</td>
</tr>
<tr>
<td>tan</td>
<td>tanf</td>
<td>(x)</td>
<td>Returns the tangent of x</td>
</tr>
<tr>
<td>tanh</td>
<td>tanhf</td>
<td>(x)</td>
<td>Returns the hyperbolic tangent of x</td>
</tr>
<tr>
<td>x**y</td>
<td></td>
<td>(x,y)</td>
<td>Returns x raised to the power y</td>
</tr>
</tbody>
</table>

The following example shows the XL Fortran interface declaration for the rsqrt scalar function:

```fortran
interface
    real*8 function rsqrt (%val(x))
    real*8 x ! Returns the reciprocal of the square root of x.
end function rsqrt
end interface
```

Notes:
- The trigonometric functions (sin, cos, tan) return NaN (Not-a-Number) for large arguments (where the absolute value is greater than 2^50*pi).
- In some cases, the MASS functions are not as accurate as the libm.a library, and they might handle edge cases differently (sqrt(Inf), for example).
- See the Mathematical Acceleration Subsystem Web site for accuracy comparisons with libm.a.

Related external information

Mathematical Acceleration Subsystem Web site, available at
http://www.ibm.com/software/awdtools/mass/

Using the vector libraries
When you compile programs with any of the following options:
- -qhot -qnostrict
the compiler automatically attempts to vectorize calls to system math functions by calling the equivalent MASS vector functions (with the exceptions of functions vatan2, vsatan2, vdnint, vdint, vsincos, vssincos, vcosisin, vscosisin, vqdr, vsqdr, vrqdr, vsrqdr, vpapcnt4, and vpapcnt8). For automatic vectorization, the compiler uses versions of the MASS functions contained in the system library libxlopt.a. You do not need to add any special calls to the MASS functions in your code, or to link to the libxlopt library.

If you are not using any of the optimization options listed above, and want to explicitly call any of the MASS vector functions, you can do so by including the XL Fortran massv.include file in your source files and linking your application with the appropriate vector library. (Information on linking is provided in “Compiling and linking a program with MASS” on page 52.)

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 a PPC970 machine, this library is the recommended choice.

libmassvp5.a
Contains some functions that have been tuned for the POWER5 architecture. The remaining functions are identical to those in libmassv.a.

libmassvp6.a
Contains some functions that have been tuned for the POWER6™ architecture. The remaining functions are identical to those in libmassv.a.

All libraries can be used in either 32-bit or 64-bit mode.

The single-precision and double-precision floating-point functions contained in the vector libraries are summarized in Table 13 on page 49. The integer functions contained in the vector libraries are summarized in Table 14 on page 50.

With the exception of a few functions (described below), all of the floating-point functions in the vector libraries accept three arguments:

- A double-precision (for double-precision functions) or single-precision (for single-precision functions) vector output argument.
- A double-precision (for double-precision functions) or single-precision (for single-precision functions) vector input argument.
- An integer vector-length argument.

The functions are of the form

\[ function\_name \ (y, x, n) \]
where \( y \) is the target vector, \( x \) is the source vector, and \( n \) is the vector length. The arguments \( y \) and \( x \) are assumed to be double-precision for functions with the prefix \( v \), and single-precision for functions with the prefix \( vs \). As examples, the following code:

```fortran
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 \( \exp(x(i)) \), where \( i=1,...,500 \).

The functions \( \text{vdiv}, \text{vsincos}, \text{vpow}, \) and \( \text{vatan2} \) (and their single-precision versions, \( \text{vsdiv}, \text{vssincos}, \text{vspow}, \) and \( \text{vssatan2} \)) take four parameters. The functions \( \text{vdiv}, \text{vpow}, \) and \( \text{vatan2} \) take the parameters \((z,x,y,n)\). The function \( \text{vdiv} \) outputs a vector \( z \) whose elements are \( x(i)/y(i) \), where \( i=1,...,n \). The function \( \text{vpow} \) outputs a vector \( z \) whose elements are \( x(i)^{y(i)} \), where \( i=1,...,n \). The function \( \text{vatan2} \) outputs a vector \( z \) whose elements are \( \arctan(x(i)/y(i)) \), where \( i=1,...,n \). The function \( \text{vsincos} \) takes the parameters \((y,z,x,n)\), and outputs two vectors, \( y \) and \( z \), whose elements are \( \sin(x(i)) \) and \( \cos(x(i)) \), respectively.

In \( \text{vcosisin}(y,x,n) \) and \( \text{vsicosisin}(y,x,n) \), \( x \) is a vector of \( n \) elements and the function outputs a vector \( y \) of \( n \) complex*16 elements of the form \((\cos(x(i)),\sin(x(i)))\).

### Table 13. MASS floating-point vector library functions

<table>
<thead>
<tr>
<th>Double-precision function</th>
<th>Single-precision function</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vacos</td>
<td>vsacos</td>
<td>((y,x,n))</td>
<td>Sets ( y(i) ) to the arc cosine of ( x(i) ), for ( i=1,...,n )</td>
</tr>
<tr>
<td>vacosh</td>
<td>vsacosh</td>
<td>((y,x,n))</td>
<td>Sets ( y(i) ) to the hyperbolic arc cosine of ( x(i) ), for ( i=1,...,n )</td>
</tr>
<tr>
<td>vasin</td>
<td>vsasin</td>
<td>((y,x,n))</td>
<td>Sets ( y(i) ) to the arc sine of ( x(i) ), for ( i=1,...,n )</td>
</tr>
<tr>
<td>vashinh</td>
<td>vsasinh</td>
<td>((y,x,n))</td>
<td>Sets ( y(i) ) to the arc hyperbolic sine of ( x(i) ), for ( i=1,...,n )</td>
</tr>
<tr>
<td>vatan2</td>
<td>vsatan2</td>
<td>((z,x,y,n))</td>
<td>Sets ( z(i) ) to the arc tangent of ( x(i)/y(i) ), for ( i=1,...,n )</td>
</tr>
<tr>
<td>vatanh</td>
<td>vsatanh</td>
<td>((y,x,n))</td>
<td>Sets ( z(i) ) to the arc hyperbolic tangent of ( x(i) ), for ( i=1,...,n )</td>
</tr>
<tr>
<td>vcbrt</td>
<td>vscbrt</td>
<td>((y,x,n))</td>
<td>Sets ( y(i) ) to the cube root of ( x(i) ), for ( i=1,...,n )</td>
</tr>
<tr>
<td>vcos</td>
<td>vsicos</td>
<td>((y,x,n))</td>
<td>Sets ( y(i) ) to the cosine of ( x(i) ), for ( i=1,...,n )</td>
</tr>
<tr>
<td>vcoh</td>
<td>vsicos</td>
<td>((y,x,n))</td>
<td>Sets ( y(i) ) to the hyperbolic cosine of ( x(i) ), for ( i=1,...,n )</td>
</tr>
<tr>
<td>vicosisin</td>
<td>vsicosisin</td>
<td>((y,x,n))</td>
<td>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 )</td>
</tr>
<tr>
<td>vdnint</td>
<td>vsdnint</td>
<td>((y,x,n))</td>
<td>Sets ( z(i) ) to the integer truncation of ( x(i) ), for ( i=1,...,n )</td>
</tr>
<tr>
<td>vexp</td>
<td>vsexp</td>
<td>((y,x,n))</td>
<td>Sets ( y(i) ) to the nearest integer to ( x(i) ), for ( i=1,...,n )</td>
</tr>
<tr>
<td>vexpm1</td>
<td>vsexpm1</td>
<td>((y,x,n))</td>
<td>Sets ( y(i) ) to ((\exp(x(i))-1), for ( i=1,...,n )</td>
</tr>
<tr>
<td>vlog</td>
<td>vslog</td>
<td>((y,x,n))</td>
<td>Sets ( y(i) ) to the natural logarithm of ( x(i) ), for ( i=1,...,n )</td>
</tr>
<tr>
<td>vlog10</td>
<td>vslog10</td>
<td>((y,x,n))</td>
<td>Sets ( y(i) ) to the base-10 logarithm of ( x(i) ), for ( i=1,...,n )</td>
</tr>
</tbody>
</table>
Table 13. MASS floating-point vector library functions (continued)

<table>
<thead>
<tr>
<th>Double-precision function</th>
<th>Single-precision function</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vlog1p</td>
<td>vslog1p</td>
<td>(y,x,n)</td>
<td>Sets y(i) to the natural logarithm of (x(i)+1), for i=1,...,n</td>
</tr>
<tr>
<td>vpow</td>
<td>vspsow</td>
<td>(z,x,y,n)</td>
<td>Sets z(i) to x(i) raised to the power y(i), for i=1,...,n</td>
</tr>
<tr>
<td>vqdrf</td>
<td>vsqdrf</td>
<td>(y,x,n)</td>
<td>Sets y(i) to the 4th root of x(i), for i=1,...,n</td>
</tr>
<tr>
<td>vrcbrt</td>
<td>vsrcbrt</td>
<td>(y,x,n)</td>
<td>Sets y(i) to the reciprocal of the cube root of x(i), for i=1,...,n</td>
</tr>
<tr>
<td>vrec</td>
<td>vsrec</td>
<td>(y,x,n)</td>
<td>Sets y(i) to the reciprocal of x(i), for i=1,...,n</td>
</tr>
<tr>
<td>vrqdrf</td>
<td>vsrqdrf</td>
<td>(y,x,n)</td>
<td>Sets y(i) to the reciprocal of the square root of x(i), for i=1,...,n</td>
</tr>
<tr>
<td>vrsqrt</td>
<td>vsrsqrt</td>
<td>(y,x,n)</td>
<td>Sets y(i) to the square root of x(i), for i=1,...,n</td>
</tr>
<tr>
<td>vsin</td>
<td>vssin</td>
<td>(y,x,n)</td>
<td>Sets y(i) to the sine of x(i), for i=1,...,n</td>
</tr>
<tr>
<td>vsincos</td>
<td>vssincos</td>
<td>(y,z,x,n)</td>
<td>Sets y(i) to the sine of x(i) and z(i) to the cosine of x(i), for i=1,...,n</td>
</tr>
<tr>
<td>vsinh</td>
<td>vssinh</td>
<td>(y,x,n)</td>
<td>Sets y(i) to the hyperbolic sine of x(i), for i=1,...,n</td>
</tr>
<tr>
<td>vsqrt</td>
<td>vssqrt</td>
<td>(y,x,n)</td>
<td>Sets y(i) to the square root of x(i), for i=1,...,n</td>
</tr>
<tr>
<td>vtan</td>
<td>vstan</td>
<td>(y,x,n)</td>
<td>Sets y(i) to the tangent of x(i), for i=1,...,n</td>
</tr>
<tr>
<td>vtanh</td>
<td>vstanh</td>
<td>(y,x,n)</td>
<td>Sets y(i) to the hyperbolic tangent of x(i), for i=1,...,n</td>
</tr>
</tbody>
</table>

Integer functions are of the form function name (x, n), where x is a vector of 4-byte (for vpopcnt4) or 8-byte (for vpopcnt8) numeric objects (integer or floating-point), and n is the vector length.

Table 14. MASS integer vector library functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>vpopcnt4</td>
<td>Returns the total number of 1 bits in the concatenation of the binary representation of x(i), for i=1,...,n, where x is vector of 32-bit objects</td>
<td>integer<em>4 function vpopcnt4 (x, n) integer</em>4 x(*) , n</td>
</tr>
<tr>
<td>vpopcnt8</td>
<td>Returns the total number of 1 bits in the concatenation of the binary representation of x(i), for i=1,...,n, where x is vector of 64-bit objects</td>
<td>integer<em>4 function vpopcnt8 (x, n) integer</em>8 x(<em>) integer</em>4 n</td>
</tr>
</tbody>
</table>

The following example shows XL Fortran interface declarations for some of the MASS double-precision vector routines:

```
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 vsqrt (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 vsqrt
end interface
```
The following example shows XL Fortran interface declarations for some of the MASS single-precision vector routines:

```
interface

subroutine vsqrt (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 vsqrt

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
```

**Overlap of input and output vectors**

In most applications, the MASS vector functions are called with disjoint input and output vectors; that is, the two vectors do not overlap in memory. Another common usage scenario is to call them with the same vector for both input and output parameters (for example, `vsin (y, y, n)`). For other kinds of overlap, be sure to observe the following restrictions, to ensure correct operation of your application:

- For calls to vector functions that take one input and one output vector (for example, `vsin (y, x, n)`):
  - The vectors `x(1:n)` and `y(1:n)` must be either disjoint or identical, or the address of `x(1)` must be greater than the address of `y(1)`. That is, if `x` and `y` are not the same vector, the address of `y(1)` must not fall within the range of addresses spanned by `x(1:n)`, or unexpected results may be obtained.

- For calls to vector functions that take two input vectors (for example, `vatan2 (y, x1, x2, n)`):
  - The previous restriction applies to both pairs of vectors `y,x1` and `y,x2`. That is, if `y` is not the same vector as `x1`, the address of `y(1)` must not fall within the range of addresses spanned by `x1(1:n)`; if `y` is not the same vector as `x2`, the address of `y(1)` must not fall within the range of addresses spanned by `x2(1:n)`.

- For calls to vector functions that take two output vectors (for example, `vsincos (y1, y2, x, n)`):
  - The above restriction applies to both pairs of vectors `y1,x` and `y2,x`. That is, if `y1` and `x` are not the same vector, the address of `y1(1)` must not fall within the range of addresses spanned by `x(1:n)`; if `y2` and `x` are not the same vector, the address of `y2(1)` must not fall within the range of addresses spanned by `x(1:n)`. Also, the vectors `y1(1:n)` and `y2(1:n)` must be disjoint.

**Consistency of MASS vector functions**

The accuracy of the vector functions is comparable to that of the corresponding scalar functions in `libmass.a`, though results might not be bitwise-identical.

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.
The following functions are consistent in all versions of the library: vcbrt, vsbrt, vrcbrt, vsrbrt, vlog, vsin, vssin, vsin, vcos, vsccos, vsexp, vacos, vsacos, vrqdr, vsqdr, vsrqdr, vacosh, vsacosh, vasin, vsasin, vstanh, vsatanh, vtan, vsanh. The following functions are consistent in libmassvp3.a, libmassvp4.a, libmassvp5.a, and libmassvp6.a: vsqrt, vrsqrt. The following functions are consistent in libmassvp4.a, libmassvp5.a, and libmassvp6.a: vrec, vsrec, vdiv, vsdiv, vexp. The following function is consistent in libmassv.a, libmassvp5.a, and libmassvp6.a: vsrsqrt. Older, inconsistent versions of some of these functions are available on the Mathematical Acceleration Subsystem for AIX Web site. If consistency is not required, there may be a performance advantage to using the older versions. For more information on consistency and avoiding inconsistency with the vector libraries, as well as performance and accuracy data, see the Mathematical Acceleration Subsystem Web site.

Related external information

- [Mathematical Acceleration Subsystem Web site, available at](http://www.ibm.com/software/awdtools/mass/)

**Compiling and linking a program with MASS**

To compile an application that calls the functions in the MASS libraries, specify `mass` and `massv` (or `massvp3`, `massvp4`, `massvp5`, or `massvp6`) on the `-l` linker option.

For example, if the MASS libraries are installed in the default directory, you could specify:

```bash
xlf prog.f -o prog -lmass -lmassv
```

The MASS functions must run in the default rounding mode and floating-point exception trapping settings.

**Using libmass.a with the math system library**

If you wish to use the libmass.a scalar library for some functions and the normal math library libm.a for other 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 Fortran program sample.f, create a file called fasttan.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:

   ```bash
   ld -export:fasttan.exp -o fasttan.o -bnoentry -lmass -bmotype:SRE
   ```

3. Archive the shared object into a library with the `ar` command. For example:

   ```bash
   ar -q libfasttan.a fasttan.o
   ```

4. Create the final executable using `xlf`, specifying the object file containing the MASS functions before the standard math library, `libm.a`. 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 math library. For example:

   ```bash
   xlf sample.f -o sample -Ldir_containing_libfasttan -lfasttan -lm
   ```
Using the Basic Linear Algebra Subprograms – BLAS

Four Basic Linear Algebra Subprograms (BLAS) functions are shipped with XL Fortran in the libxlopt library.

The functions consist of the following:
- SGEMV (single-precision) and DGEMV (double-precision), which compute the matrix-vector product for a general matrix or its transpose
- SGEMM (single-precision) and DGEMM (double-precision), which perform combined matrix multiplication and addition for general matrices or their transposes

Note: Some error-handling code has been removed from the BLAS functions in libxlopt, and no error messages are emitted for calls to these functions.

“BLAS function syntax” describes the interfaces for the XL Fortran BLAS functions, which are similar to those of the equivalent BLAS functions shipped in IBM’s Engineering and Scientific Subroutine Library (ESL); for more detailed information and examples of usage of these functions, you may wish to consult the Engineering and Scientific Subroutine Library Guide and Reference, available at http://publib.boulder.ibm.com/clresctr/windows/public/esslbooks.html.

“Linking the libxlopt library” on page 55 describes how to link to the XL Fortran libxlopt library if you are also using a third-party BLAS library.

BLAS function syntax

The interfaces for the SGEMV and DGEMV functions are as follows:

CALL SGEMV(trans, m, n, alpha, a, lda, x, incx, beta, y, incy)
CALL DGEMV(trans, m, n, alpha, a, lda, x, incx, beta, y, incy)

The parameters are as follows:

trans
  is a single character indicating the form of the input matrix a, where:
  - 'N' or 'n' indicates that a is to be used in the computation
  - 'T' or 't' indicates that the transpose of a is to be used in the computation

m
  represents:
  - the number of rows in input matrix a
  - the length of vector y, if 'N' or 'n' is used for the trans parameter
  - the length of vector x, if 'T' or 't' is used for the trans parameter

The number of rows must be greater than or equal to zero, and less than the leading dimension of the matrix a (specified in lda)

n
  represents:
  - the number of columns in input matrix a
  - the length of vector x, if 'N' or 'n' is used for the trans parameter

Note: The MASS sincos function is automatically linked if you export MASS cosisin; MASS cos function is automatically linked if you export MASS sin; MASS atan2 is automatically linked if you export MASS atan.

Related external information
- ar and ld in the AIX Commands Reference, Volumes 1 - 6

Chapter 6. High performance libraries  53
• the length of vector \( y \), if 'T' or 't' is used for the \( \text{trans} \) parameter

The number of columns must be greater than or equal to zero.

\[ \text{alpha} \]

is the scaling constant \( \alpha \)

\[ \text{a} \]

is the input matrix of single-precision (for SGEMV) or double-precision (for DGEMV) real values

\[ \text{lda} \]

is the leading dimension of the array specified by \( a \). The leading dimension must be greater than zero. The leading dimension must be greater than or equal to 1 and greater than or equal to the value specified in \( m \).

\[ \text{x} \]

is the input vector of single-precision (for SGEMV) or double-precision (for DGEMV) real values.

\[ \text{incy} \]

is the stride for vector \( x \). It can have any value.

\[ \text{beta} \]

is the scaling constant \( \beta \)

\[ \text{y} \]

is the output vector of single-precision (for SGEMV) or double-precision (for DGEMV) real values.

\[ \text{incy} \]

is the stride for vector \( y \). It must not be zero.

\textbf{Note:} Vector \( y \) must have no common elements with matrix \( a \) or vector \( x \); otherwise, the results are unpredictable.

The prototypes for the SGEMM and DGEMM functions are as follows:

\[
\text{CALL SGEMM}(\text{transa}, \text{transb}, l, n, m, \text{alpha}, a, \text{lda}, b, \text{ldb}, \text{beta}, c, \text{ldc})
\]

\[
\text{CALL DGEMM}(\text{transa}, \text{transb}, l, n, m, \text{alpha}, a, \text{lda}, b, \text{ldb}, \text{beta}, c, \text{ldc})
\]

The parameters are as follows:

\[ \text{transa} \]

is a single character indicating the form of the input matrix \( a \), where:

• 'N' or 'n' indicates that \( a \) is to be used in the computation
• 'T' or 't' indicates that the transpose of \( a \) is to be used in the computation

\[ \text{transb} \]

is a single character indicating the form of the input matrix \( b \), where:

• 'N' or 'n' indicates that \( b \) is to be used in the computation
• 'T' or 't' indicates that the transpose of \( b \) is to be used in the computation

\[ l \]

represents the number of rows in output matrix \( c \). The number of rows must be greater than or equal to zero, and less than the leading dimension of \( c \).

\[ n \]

represents the number of columns in output matrix \( c \). The number of columns must be greater than or equal to zero.

\[ m \]

represents:

• the number of columns in matrix \( a \), if 'N' or 'n' is used for the \( \text{transa} \) parameter
• the number of rows in matrix \( a \), if 'T' or 't' is used for the \( \text{transa} \) parameter

and:
• the number of rows in matrix $b$, if 'N' or 'n' is used for the $transb$ parameter
• the number of columns in matrix $b$, if 'T' or 't' is used for the $transb$ parameter

$m$ must be greater than or equal to zero.

$\alpha$

is the scaling constant $\alpha$

$a$

is the input matrix $a$ of single-precision (for SGEMM) or double-precision (for DGEMM) real values

$lda$ is the leading dimension of the array specified by $a$. The leading dimension must be greater than zero. If $transa$ is specified as 'N' or 'n', the leading dimension must be greater than or equal to 1. If $transa$ is specified as 'T' or 't', the leading dimension must be greater than or equal to the value specified in $m$.

$b$

is the input matrix $b$ of single-precision (for SGEMM) or double-precision (for DGEMM) real values.

$ldb$ is the leading dimension of the array specified by $b$. The leading dimension must be greater than zero. If $transb$ is specified as 'N' or 'n', the leading dimension must be greater than or equal to the value specified in $m$. If $transa$ is specified as 'T' or 't', the leading dimension must be greater than or equal to the value specified in $n$.

$\beta$

is the scaling constant $\beta$

$c$

is the output matrix $c$ of single-precision (for SGEMM) or double-precision (for DGEMM) real values.

$ldc$ is the leading dimension of the array specified by $c$. The leading dimension must be greater than zero. If $transb$ is specified as 'N' or 'n', the leading dimension must be greater than or equal to 0 and greater than or equal to the value specified in $l$.

Note: Matrix $c$ must have no common elements with matrices $a$ or $b$; otherwise, the results are unpredictable.

**Linking the libxlopt library**

By default, the libxlopt library is linked with any application you compile with XL Fortran. However, if you are using a third-party BLAS library, but want to use the BLAS routines shipped with libxlopt, you must specify the libxlopt library before any other BLAS library on the command line at link time. For example, if your other BLAS library is called libblas, you would compile your code with the following command:

`xlf app.f -lxlopt -lblas`

The compiler will call the SGEMV, DGEMV, SGEMM, and DGEMM functions from the libxlopt library, and all other BLAS functions in the libblas library.
Chapter 7. Parallel programming with XL Fortran

Parallel programming with XL Fortran involves a combination of compiling, setting of runtime options, and optimization of your code, by incorporating SMP directives and by using the pthreads library module.

XL Fortran supports both the OpenMP specification, as understood and interpreted by IBM as well as the POSIX 1003.1-1996 standard, and the Draft 7 POSIX pthreads API on AIX.

Compiling your SMP code

To compile SMP code, you must specify the -qsmp compiler option. When compiling with -qsmp, the driver links the libraries found on the smplibraries line in the active stanza of your configuration file.

If you specify -qsmp, you must use an SMP appropriate invocation command. Use any of the following invocations to compile SMP code or to ensure that the compiler links threadsafe libraries:

- xlf_r
- xlf_r7
- xlf90_r
- xlf90_r7
- xlf95_r
- xlf95_r7
- xlf2003_r

For information on linking your 32- and 64-bit SMP code, see Linking 32-bit and Linking 64-bit SMP object files in the XL Fortran Compiler Reference.

Related reference

See -qsmp in the Compiler Reference

The _OPENMP C preprocessor macro and conditional compilation

You can use sentinels to mark specific lines of an XL Fortran program for conditional compilation. This allows you to port code that contains statements that are only valid or applicable in an SMP environment to a non-SMP environment.

You can do this using conditional compilation lines, or the _OPENMP C preprocessor macro. This macro is defined when the C preprocessor is invoked and you specify the -qsmp=omp compiler option. See Passing Fortran files through the C preprocessor in the Editing, Compiling, Linking, and Running XL Fortran Programs section of the XL Fortran Compiler Reference for an example of using this macro.

The following example uses conditional compilation lines to hide OpenMP runtime routines. You cannot easily compile code that calls OpenMP runtime routines in a non-OpenMP environment without using conditional compilation. Since calls to the runtime routines are not directives, they cannot be hidden by the !$OMP trigger. If you do not compile the example with -qsmp=omp the variable that stores the number of threads is assigned the value of 8.

Example of conditional compilation lines
PROGRAM PAR_MAT_MUL
IMPLICIT NONE
INTEGER(KIND=8) :: I, J, NTHREADS
INTEGER(KIND=8), PARAMETER :: N = 60
INTEGER(KIND=8), DIMENSION(N, N) :: A(I, J), C(I, J)
INTEGER(KIND=8) :: SUMI
!
!$ INTEGER OMP_GET_NUM_THREADS
COMMON/DATA/ A(I, J), C(I, J)
!$OMP THREADPRIVATE (/DATA/)
!$OMP PARALLEL
  FORALL(I=1:N, J=1:N) A(I, J) = (I-N/2)**2 + (J+N/2)
  FORALL(I=1:N, J=1:N) B(I, J) = 3 - ((I/2) + (J-N/2)**2)
!$OMP MASTER
  NTHREADS = 8
!$ NTHREADS = OMP_GET_NUM_THREADS()
!$OMP END MASTER
!$OMP END PARALLEL
!
!$OMP PARALLEL DEFAULT(PRIVATE), COPYIN(A(I, J), C(I, J)), SHARED(NTHREADS)
!$OMP DO
  DO J = 1, NTHREADS
    CALL IMAT_MUL(SUM)
  ENDDO
!$OMP END DO
!$OMP END PARALLEL
END

For information on using sentinels, see Conditional compilation in the XL Fortran Language Reference.

Setting OMP and SMP run time options

Set the necessary XLSMPOPTS environment variables, and the OpenMP environment variables when you write parallel code.

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 runtime 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 runtime 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 runtime library performs an initial division of the iterations into 
\( \text{number_of_threads} \) partitions. The number of iterations that these partitions contain is:

\[
\text{CEILING}(\text{number_of_iterations} / \text{number_of_threads})
\]

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

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

**affinity[=n]**

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

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

**dynamic[=n]**

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

**guided[=n]**

The iterations of a loop are divided into progressively smaller chunks until a minimum chunk size of \( n \) loop iterations is reached. If you have not specified \( n \), the default value for \( n \) is 1 iteration.

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

**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. For more information, see the description of the schedule directive in the XL Fortran Language Reference.

Parallel execution options

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

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

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

usrthds$=num
Specifies the maximum number of threads ($num$) that you expect your code will explicitly create if the code does explicit thread creation. The default value for $num$ is 0. For more information, see the [NUM_PARTHDS] intrinsic function in the XL Fortran Language Reference.

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

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

stackcheck$=[num] | nostackcheck
When -qsmp=stackcheck is in effect, enables stack overflow checking for worker threads at runtime. $num$ is the size of the stack in bytes; when the remaining stack size is less than this value, a runtime warning message is issued. If you do not specify a value for $num$, the default value is 4096 bytes. Note that this option only has an effect when -qsmp=stackcheck has also been specified at compile time. See -qsmp in the XL Fortran Compiler Reference for more information.

startproc$=cpu_id
Enables thread binding and specifies the CPU ID to which the first thread binds. If the value provided is outside the range of available processors, the SMP run time issues a warning message and no threads are bound.
**stride=num**

Specifies the increment used to determine the CPU ID to which subsequent threads bind. *num* must be greater than or equal to 1. If the value provided causes a thread to bind to a CPU outside the range of available processors, a warning message is issued and no threads are bound.

**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 **xlsmpr** runtime 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:
parths=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 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 nth 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 runtime 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 FALSE. 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 runtime environment might deploy extra threads to form the team of threads for the nested parallel region. If you set this environment variable to FALSE, nested parallelism is disabled.

The default value for OMP_NESTED is FALSE.

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

Currently, XL Fortran does not support OpenMP nested parallelism.

**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:
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.

If the number of threads you request exceeds the number your execution environment can support, your application will terminate.

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(Its/Iters)
```

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(Its/Iters))
```

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:

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

## Optimizing your SMP code

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.

<table>
<thead>
<tr>
<th>Commonly used <code>-qsmp</code> suboptions</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>auto</td>
<td>Instructs the compiler to automatically generate parallel code where possible without user assistance. This option also recognizes all the SMP directives.</td>
</tr>
<tr>
<td>omp</td>
<td>Enforces compliance with the OpenMP API for specifying explicit parallelism.</td>
</tr>
<tr>
<td>opt</td>
<td>Instructs the compiler to optimize as well as parallelize. The optimization is equivalent to <code>-O2</code> <code>-qhot</code> in the absence of other optimization options. The default setting of <code>-qsmp</code> is <code>-qsmp=auto:nompp:opt</code>.</td>
</tr>
<tr>
<td>suboptions</td>
<td>Other values for the suboption provide control over thread scheduling, nested parallelism, locking, and so on.</td>
</tr>
</tbody>
</table>

Certain thread environment variables like MALLOCMULTIHEAP, SPINLOOPTIME, or YIELDLOOPTIME may improve application performance, as well. For more information on them, see the information in the AIX Information Center at URL: [http://publib.boulder.ibm.com/infocenter/pseries/v5r3/index.jsp](http://publib.boulder.ibm.com/infocenter/pseries/v5r3/index.jsp)

## Developing and running SMP applications

- By default, the parallelization performed is both user-directed and automatic. Use `-qsmp=omp:nompp:opt` if you are compiling an OpenMP program and do not want automatic parallelization.
- 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.
An introduction to SMP directives

The SMP directives allow you to exert control over parallelization. For example, the PARALLEL DO directive specifies that the loop immediately following the directive should be run in parallel. All SMP directives are comment form directives.

For more information on rules and syntax for comment form directives, see Comment and noncomment form directives in the XL Fortran Language Reference.

XL Fortran supports a number of SMP directives, divided as follows. To ensure the greatest portability of code, OpenMP directives are recommended where possible. Use the OpenMP trigger constant,OMP for OpenMP directives, but do not use this trigger constant with any other directive. OpenMP directives must not appear in PURE and ELEMENTAL procedures.

Parallel region construct

Parallel constructs form the foundation of OpenMP based parallel execution in XL Fortran. The PARALLEL/END PARALLEL directive pair forms a basic parallel construct. Each time an executing thread enters a parallel region, it creates a team of threads and becomes master of that team. This allows parallel execution to take place within that construct by the threads in that team. The following directives are necessary for a parallel region:

<table>
<thead>
<tr>
<th>PARALLEL</th>
<th>END PARALLEL</th>
</tr>
</thead>
</table>

Work-sharing constructs

Work-sharing constructs divide the execution of code enclosed by the construct between threads in a team. For work-sharing to take place, the construct must be enclosed within the dynamic extent of a parallel region. For further information on work-sharing constructs, see the following directives:

<table>
<thead>
<tr>
<th>DO</th>
<th>END DO</th>
</tr>
</thead>
<tbody>
<tr>
<td>SECTIONS</td>
<td>END SECTIONS</td>
</tr>
<tr>
<td>WORKSHARE</td>
<td>END WORKSHARE</td>
</tr>
</tbody>
</table>

Combined parallel work-sharing constructs

A combined parallel work-sharing construct allows you to specify a parallel region that already contains a single work-sharing construct. These combined constructs are semantically identical to specifying a parallel construct enclosing a single work-sharing construct. For more information on implementing combined constructs, see the following directives:

<table>
<thead>
<tr>
<th>PARALLEL DO</th>
<th>END PARALLEL DO</th>
</tr>
</thead>
<tbody>
<tr>
<td>PARALLEL SECTIONS</td>
<td>END PARALLEL SECTIONS</td>
</tr>
<tr>
<td>PARALLEL WORKSHARE</td>
<td>END PARALLEL WORKSHARE</td>
</tr>
</tbody>
</table>

Synchronization constructs

The following directives allow you to synchronize the execution of a parallel region by multiple threads in a team:
Other OpenMP Directives
The following OpenMP directives provide additional SMP functionality:

- MASTER
- END MASTER
- TASK
- THREADPRIVATE

Non-OpenMP SMP directives
The following directives provide additional SMP functionality:

- DO SERIAL
- SCHEDULE
- THREADLOCAL

Detailed descriptions of SMP directives
See the alphabetical list of all SMP directives supported by XL Fortran.

For information on directive clauses, see “OpenMP directive clauses” on page 115.

**ATOMIC**

**Purpose**

You can use the **ATOMIC** directive to update a specific memory location safely within a parallel region. When you use the **ATOMIC** directive, you ensure that only one thread is writing to the memory location at a time, avoiding errors which might occur from simultaneous writes to the same memory location.

Normally, you would protect a shared variable within a **CRITICAL** construct if it is being updated by more than one thread at a time. However, certain platforms support atomic operations for updating variables. For example, some platforms might support a hardware instruction that reads from a memory location, calculates something and writes back to the location all in one atomic action. The **ATOMIC** directive instructs the compiler to use an atomic operation whenever possible. Otherwise, the compiler will use some other mechanisms to perform an atomic update.

The **ATOMIC** directive only takes effect if you specify the `-qsmp` compiler option.
Syntax

where atomic_statement is:

update_variable = update_variable operator expression

operator          is one of +, −, *, /, .AND., .OR., .EQV., .NEQV. or .XOR.

expression        is a scalar expression that does not reference update_variable.

Rules

The ATOMIC directive applies only to the statement which immediately follows it.

The expression in an atomic_statement is not evaluated atomically. You must ensure that no race conditions exist in the calculation.

All references made using the ATOMIC directive to the storage location of an update_variable within the entire program must have the same type and type parameters.

The function intrinsic, the operator operator, and the assignment must be the intrinsic function, operator and assignment and not a redefined intrinsic function, defined operator or defined assignment.

Examples

Example 1: In the following example, multiple threads are updating a counter. ATOMIC is used to ensure that no updates are lost.

PROGRAM P
  R = 0.0
!$OMP PARALLEL DO SHARED(R)
DO I=1, 10
!$OMP ATOMIC
  R = R + 1.0
END DO
PRINT *, R
END PROGRAM P
Expected output:
10.0

**Example 2:** In the following example, an **ATOMIC** directive is required, because it is uncertain which element of array Y will be updated in each iteration.

```fortran
PROGRAM P
  INTEGER, DIMENSION(10) :: Y, INDEX
  INTEGER B
  Y = 5
  READ(*,*) INDEX, B
  !$OMP PARALLEL DO SHARED(Y)
  DO I = 1, 10
    !$OMP ATOMIC
    Y(INDEX(I)) = MIN(Y(INDEX(I)), B)
  END DO
  PRINT *, Y
END PROGRAM P
```

Input data:
10 10 8 8 6 6 4 4 2 2 4

Expected output:
5 4 5 4 5 4 5 4 4

**Example 3:** The following example is invalid, because you cannot use an **ATOMIC** operation to reference an array.

```fortran
PROGRAM P
  REAL ARRAY(10)
  ARRAY = 0.0
  !$OMP PARALLEL DO SHARED(ARRAY)
  DO I = 1, 10
    !$OMP ATOMIC
    ARRAY = ARRAY + 1.0
  END DO
  PRINT *, ARRAY
END PROGRAM P
```

**Example 4:** The following example is invalid. The *expression* must not reference the *update_variable*.

```fortran
PROGRAM P
  R = 0.0
  !$OMP PARALLEL DO SHARED(R)
  DO I = 1, 10
    !$OMP ATOMIC
    R = R + R
  END DO
  PRINT *, R
END PROGRAM P
```

Related reference

“**CRITICAL / END CRITICAL**” on page 71
“**PARALLEL / END PARALLEL**” on page 83

See `-qsmp` in the Compiler Reference
**BARRIER**

**Purpose**

The BARRIER directive enables you to synchronize all threads in a team. When a thread encounters a BARRIER directive, it will wait until all other threads in the team reach the same point.

**Type**

The BARRIER directive only takes effect if you specify the -qsmp compiler option.

**Syntax**

```
BARRIER
```

**Rules**

A BARRIER directive binds to the closest dynamically enclosing PARALLEL directive, if one exists.

A BARRIER directive cannot appear within the dynamic extent of the CRITICAL, DO (work-sharing), MASTER, ORDERED, PARALLEL DO, PARALLEL SECTIONS, PARALLEL WORKSHARE, SECTIONS, SINGLE, TASK, and WORKSHARE directives.

All threads in the team must encounter the BARRIER directive if any thread encounters it.

All BARRIER directives and work-sharing constructs must be encountered in the same order by all threads in the team.

In addition to synchronizing the threads in a team, the BARRIER directive implies the FLUSH directive without the variable_name_list.

**Examples**

**Example 1:** An example of the BARRIER directive binding to the PARALLEL directive. Note: To calculate C, we need to ensure that A and B have been completely assigned to, so threads need to wait.

```fortran
SUBROUTINE SUB1
  INTEGER A(1000), B(1000), C(1000)
!$OMP PARALLEL
!$OMP DO
  DO I = 1, 1000
    A(I) = SIN(I*2.5)
  END DO
!$OMP END DO NOWAIT
!$OMP DO
  DO J = 1, 10000
    B(J) = X + COS(J*5.5)
  END DO
!$OMP END DO NOWAIT
  ...
```

XL Fortran: Optimization and Programming Guide
Example 2: An example of a BARRIER directive that incorrectly appears inside a CRITICAL section. This can result in a deadlock because only one thread can enter a CRITICAL section at a time.

```fortran
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP CRITICAL
   DO I = 1, 10
      X = X + 1
   END DO
!$OMP END CRITICAL
!$OMP END PARALLEL
```

Related reference

“FLUSH” on page 77
“PARALLEL / END PARALLEL” on page 83

See -qsmp in the Compiler Reference

CRITICAL / END CRITICAL

Purpose

The CRITICAL construct allows you to define independent blocks of code that are to be run by at most one thread at a time. It includes a CRITICAL directive that is followed by a block of code and ends with an END CRITICAL directive.

Type

The CRITICAL and END CRITICAL directives only take effect if you specify the -qsmp compiler option.

Syntax

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CRITICAL</td>
<td>(lock_name)</td>
<td></td>
</tr>
<tr>
<td>block</td>
<td></td>
<td></td>
</tr>
<tr>
<td>END CRITICAL</td>
<td>(lock_name)</td>
<td></td>
</tr>
</tbody>
</table>
```

lock_name provides a way of distinguishing different CRITICAL constructs of code.

block represents the block of code to be executed by at most one thread at a time.
Rules

The optional lock_name is a name with global scope. You must not use the lock_name to identify any other global entity in the same executable program.

If you specify the lock_name on the CRITICAL directive, you must specify the same lock_name on the corresponding END CRITICAL directive.

If you specify the same lock_name for more than one CRITICAL construct, the compiler will allow only one thread to execute any one of these CRITICAL constructs at any one time. CRITICAL constructs that have different lock_names may be run in parallel.

The same lock protects all CRITICAL constructs that do not have an explicit lock_name. In other words, the compiler will assign the same lock_name, thereby ensuring that only one thread enters any unnamed CRITICAL construct at a time.

The lock_name must not share the same name as any local entity of Class 1.

It is illegal to branch into or out of a CRITICAL construct.

The CRITICAL construct may appear anywhere in a program.

Although it is possible to nest a CRITICAL construct within a CRITICAL construct, a deadlock situation may result. The -qsmp=rec_locks compiler option can be used to prevent deadlocks. See the XL Fortran Compiler Reference for more information. The OpenMP API does not allow nested CRITICAL constructs.

CRITICAL and END CRITICAL directives imply the FLUSH directive without the variable_name_list.

Examples

Example 1: Note that in this example the CRITICAL construct appears within a DO loop that has been marked with the PARALLEL DO directive.

EXPR=0
!OMP$ PARALLEL DO PRIVATE (I)
  DO I = 1, 100
  !OMP$ CRITICAL
    EXPR = EXPR + A(I) * I
  !OMP$ END CRITICAL
  END DO

Example 2: An example specifying a lock_name on the CRITICAL construct.

!SMP$ PARALLEL DO PRIVATE(T)
  DO I = 1, 100
    T = B(I) + B(I-1)
  !SMP$ CRITICAL (LOCK)
    SUM = SUM + T
  !SMP$ END CRITICAL (LOCK)
  END DO

Related reference

"ATOMIC" on page 67
"FLUSH" on page 77

See Global entity in the Language Reference
**DO / END DO**

**Purpose**

The **DO** (work-sharing) construct enables you to divide the execution of the loop among the members of the team that encounter it. The **END DO** directive enables you to indicate the end of a **DO** loop that is specified by the **DO** (work-sharing) directive.

The **DO** (work-sharing) and **END DO** directives only take effect when you specify the **-qsmp** compiler option.

**Syntax**

```
DO do_clause
  ...do_loop...
END DO
```

where **do_clause** is:

```
collapse_clause
  -firstprivate_clause
  -lastprivate_clause
  -ordered_clause
  -private_clause
  -reduction_clause
  -schedule_clause
```

**collapse_clause**

See — “COLLAPSE” on page 117.

**firstprivate_clause**

See — “FIRSTPRIVATE” on page 124.

**lastprivate_clause**

See — “LASTPRIVATE” on page 125.
ordered_clause
See — “ORDERED” on page 127

private_clause
See — “PRIVATE” on page 128

reduction_clause
See — “REDUCTION” on page 129

schedule_clause
See — “SCHEDULE” on page 132

Rules

The first noncomment line (not including other directives) that follows the DO (work-sharing) directive must be a DO loop. This line cannot be an infinite DO or DO WHILE loop. The DO (work-sharing) directive applies only to the DO loop that is immediately following the directive, and not to any nested DO loops, unless the COLLAPSE clause is specified.

The END DO directive is optional. If you use the END DO directive, it must immediately follow the end of the DO loop.

You may have a DO construct that contains several DO statements. If the DO statements share the same DO termination statement, and an END DO directive follows the construct, you can only specify a work-sharing DO directive for the outermost DO statement of the construct.

If you specify NOWAIT on the END DO directive, a thread that completes its iterations of the loop early will proceed to the instructions following the loop. The thread will not wait for the other threads of the team to complete the DO loop. If you do not specify NOWAIT on the END DO directive, each thread will wait for all other threads within the same team at the end of the DO loop.

If you do not specify the NOWAIT clause, the END DO directive implies the FLUSH directive without the variable_name_list.

All threads in the team must encounter the DO (work-sharing) directive if any thread encounters it. A DO loop must have the same loop boundary and step value for each thread in the team. All work-sharing constructs and BARRIER directives that are encountered must be encountered in the same order by all threads in the team.

A DO (work-sharing) directive must not appear within the dynamic extent of a CRITICAL, MASTER, or ORDERED construct. In addition, it must not appear within the dynamic extent of a PARALLEL SECTIONS construct, work-sharing construct, or PARALLEL DO loop, unless it is within the dynamic extent of a PARALLEL construct. The DO directive must not appear within a TASK region.

You cannot follow a DO (work-sharing) directive by another DO (work-sharing) directive. You can only specify one DO (work-sharing) directive for a given DO loop.

The DO (work-sharing) directive cannot appear with either an INDEPENDENT or DO SERIAL directive for a given DO loop.
Examples

Example 1: An example of several independent DO loops within a PARALLEL construct. No synchronization is performed after the first work-sharing DO loop, because NOWAIT is specified on the END DO directive.

```fortran
!$OMP PARALLEL
  !$OMP DO
    DO I = 2, N
      B(I) = (A(I) + A(I-1)) / 2.0
    END DO
  !$OMP END DO NOWAIT
  !$OMP DO
    DO J = 2, N
      C(J) = SQRT(REAL(J*J))
    END DO
  !$OMP END DO
  C(5) = C(5) + 10
!$OMP END PARALLEL
END
```

Example 2: An example of SHARED, and SCHEDULE clauses.

```fortran
!$OMP PARALLEL SHARED(A)
  !$OMP DO SCHEDULE(STATIC,10)
    DO I = 1, 1000
      A(I) = I * 4
    END DO
  !$OMP END DO
!$OMP END DO
!$OMP END PARALLEL
```

Example 3: An example of both a MASTER and a DO (work-sharing) directive that bind to the closest enclosing PARALLEL directive.

```fortran
!$OMP PARALLEL DEFAULT(PRIVATE), SHARED(X)
    Y = 100
!$OMP MASTER
    PRINT *, Y
!$OMP END MASTER
  !$OMP DO
    DO I = 1, 10
      X(I) = I
      X(I) = X(I) + Y
    END DO
!$OMP END PARALLEL
END
```

Example 4: An example of both the FIRSTPRIVATE and the LASTPRIVATE clauses on DO (work-sharing) directives.

```fortran
  X = 100
  !$OMP PARALLEL PRIVATE(I), SHARED(X,Y)
  !$OMP DO FIRSTPRIVATE(X), LASTPRIVATE(X)
    DO I = 1, 80
      Y(I) = X + I
      X = I
    END DO
!$OMP END PARALLEL
END
```

Example 6: A valid example of a work-sharing DO directive applied to nested DO statements with a common DO termination statement.

```fortran
!$OMP DO ! A work-sharing DO directive can ONLY
  ! precede the outermost DO statement.
  DO 100 I = 1,10
```
! !$OMP DO **Error** ! Placing the OMP DO directive here is
! invalid

DO 100 J= 1,10
!
... 
100 CONTINUE
!$OMP END DO

Related reference

"COLLAPSE" on page 117
See DO in the Language Reference
"DO SERIAL"
"FLUSH" on page 77
See INDEPENDENT in the Language Reference
See Loop parallelization in the Language Reference
"ORDERED / END ORDERED" on page 81
"PARALLEL / END PARALLEL" on page 83
"PARALLEL DO / END PARALLEL DO" on page 86
"PARALLEL SECTIONS / END PARALLEL SECTIONS" on page 90
"SCHEDULE" on page 94

DO SERIAL

Purpose

The DO SERIAL directive indicates to the compiler that the DO loop that is immediately following the directive must not be parallelized. This directive is useful in blocking automatic parallelization for a particular DO loop. The DO SERIAL directive only takes effect if you specify the -qsmp compiler option.

Syntax

```
* DO SERIAL
```

Rules

The first noncomment line (not including other directives) that is following the DO SERIAL directive must be a DO loop. The DO SERIAL directive applies only to the DO loop that immediately follows the directive and not to any loops that are nested within that loop.

You can only specify one DO SERIAL directive for a given DO loop. The DO SERIAL directive must not appear with the DO, or PARALLEL DO directive on the same DO loop.

White space is optional between DO and SERIAL.

You should not use the OpenMP trigger constant with this directive.
Examples

Example 1: An example with nested DO loops where the inner loop (the J loop) is not parallelized.

```fortran
!$OMP PARALLEL DO PRIVATE(S,I), SHARED(A)
  DO I=1, 500
    S=0
!SMP$ DOSERIAL
    DO J=1, 500
      S=S+1
    ENDDO
    A(I)=S+I
  ENDDO
```

Example 2: An example with the DOSERIAL directive applied in nested loops. In this case, if automatic parallelization is enabled the I or K loop may be parallelized.

```fortran
DO I=1, 100
  !SMP$ DOSERIAL
  DO J=1, 100
    DO K=1, 100
      ARR(I,J,K)=I+J+K
    ENDDO
  ENDDO
ENDDO
```

Related reference

“DO / END DO” on page 73

See DO in the Language Reference

See Loop parallelization in the Language Reference

“PARALLEL DO / END PARALLEL DO” on page 86

See -qdirective in the Compiler Reference

See -qsmp in the Compiler Reference

FLUSH

Purpose

The FLUSH directive ensures that each thread has access to data generated by other threads. This directive is required because the compiler may keep values in processor registers if a program is optimized. The FLUSH directive ensures that the memory images that each thread views are consistent.

The FLUSH directive only takes effect if you specify the -qsmp compiler option.

You might be able to improve the performance of your program by using the FLUSH directive instead of the VOLATILE attribute. The VOLATILE attribute causes variables to be flushed after every update and before every use, while FLUSH causes variables to be written to or read from memory only when specified.

Syntax
Rules

You can specify this directive anywhere in your code; however, if you specify it outside of the dynamic extent of a parallel region, it is ignored.

If you specify a variable_name_list, only the variables in that list are written to or read from memory (assuming that they have not been written or read already). All variables in the variable_name_list must be at the current scope and must be thread visible. Thread visible variables can be any of the following:

- Globally visible variables (common blocks and module data)
- Local and host-associated variables with the SAVE attribute
- Local variables without the SAVE attribute that are specified in a SHARED clause in a parallel region within the subprogram
- Local variables without the SAVE attribute that have had their addresses taken
- All pointer dereferences
- Dummy arguments

If an item or a subobject of an item in the variable_name_list has the POINTER attribute, the allocation and association status of the POINTER item is flushed, but the pointer target is not. If an item in the variable_name_list is an integer pointer, the pointer is flushed, but the object to which it points is not. If an item in the variable_name_list has the ALLOCATABLE attribute and the item is allocated, the allocated object is flushed. Otherwise, the allocation status is flushed.

If you do not specify a variable_name_list, all thread visible variables are written to or read from memory.

When a thread encounters the FLUSH directive, it writes into memory the modifications to the affected variables. The thread also reads the latest copies of the variables from memory if it has local copies of those variables: for example, if it has copies of the variables in registers.

It is not mandatory for all threads in a team to use the FLUSH directive. However, to guarantee that all thread visible variables are current, any thread that modifies a thread visible variable should use the FLUSH directive to update the value of that variable in memory. If you do not use FLUSH or one of the directives that implies FLUSH (see below), the value of the variable might not be the most recent one.

The FLUSH directive does not imply any ordering between the directive and operations on variables not in the variable_name_list. The FLUSH directive does not imply any ordering between two or more FLUSH constructs if the constructs do not have any variables in common in the variable_name_list.

Note that FLUSH is not atomic. You must FLUSH shared variables that are controlled by a shared lock variable with one directive and then FLUSH the lock variable with another. This guarantees that the shared variables are written before the lock variable.
The following directives imply a **FLUSH** directive without the *variable_name_list* unless you specify a **NOWAIT** clause for those directives to which it applies:

- **BARRIER**
- **CRITICAL/END CRITICAL**
- **END DO**
- **END SECTIONS**
- **END SINGLE**
- **PARALLEL/END PARALLEL**
- **PARALLEL DO/END PARALLEL DO**
- **PARALLEL SECTIONS/END PARALLEL SECTIONS**
- **PARALLEL WORKSHARE/END PARALLEL WORKSHARE**
- **ORDERED/END ORDERED**

The **ATOMIC** directive implies a **FLUSH** directive with the *variable_name_list*. The *variable_name_list* contains only the object updated in the **ATOMIC** construct.

The following routines imply a **FLUSH** directive without the *variable_name_list*:

- During **OMP_SET_LOCK** and **OMP_UNSET_LOCK** regions.
- During **OMP_TEST_LOCK**, **OMP_SET_NEST_LOCK** **OMP_UNSET_NEST_LOCK** and **OMP_TEST_NEST_LOCK** regions, if the region causes the lock to be set or unset.

### Examples

**Example 1:** In the following example, two threads perform calculations in parallel and are synchronized when the calculations are complete:

```fortran
PROGRAM P
  INTEGER INSYNC(0:1), IAM

  !$OMP PARALLEL DEFAULT(PRIVATE) SHARED(INSYNC)
     IAM = OMP_GET_THREAD_NUM()
     INSYNC(IAM) = 0
  !$OMP BARRIER
  CALL WORK
  !$OMP FLUSH(INSYNC)
     INSYNC(IAM) = 1  ! Each thread sets a flag once it has completed its work.
     INSYNC(1-IAM) = 0 ! One thread waits for another to complete.
  !$OMP FLUSH(INSYNC) ! its work.
  END DO
  !$OMP END PARALLEL
END PROGRAM
```

**Example 2:** The following example is not valid, because it attempts to use **FLUSH** with a variable that is not thread visible:

```fortran
END SUBROUTINE WORK
```

Chapter 7. Parallel programming with XL Fortran
FUNCTION F()
INTEGER, AUTOMATIC :: i
!$OMP FLUSH(I)
END FUNCTION F

MASTER / END MASTER

Purpose

The MASTER construct enables you to define a block of code that will be run by only the master thread of the team. It includes a MASTER directive that precedes a block of code and ends with an END MASTER directive.

Type

The MASTER and END MASTER directives only take effect if you specify the -qomp compiler option.

Syntax

```
MASTER
block
END MASTER
```

*block* represents the block of code that will be run by the master thread of the team.

Rules

It is illegal to branch into or out of a MASTER construct.

A MASTER directive binds to the closest dynamically enclosing PARALLEL directive, if one exists.

A MASTER directive cannot appear within the dynamic extent of a work-sharing construct or within the dynamic extent of the PARALLEL DO, PARALLEL SECTIONS, and PARALLEL WORKSHARE directives.

No implied barrier exists on entry to, or exit from, the MASTER construct.

Examples

Example 1: An example of the MASTER directive binding to the PARALLEL directive.

```
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP MASTER
    Y = 10.0
    X = 0.0
    DO I = 1, 4
       X = X + COS(Y) + I
    END DO
```

XL Fortran: Optimization and Programming Guide
ORDINARY / END ORDINARY

Purpose

The ORDERED / END ORDERED directives cause the iterations of a block of code within a parallel loop to be executed in the order that the loop would execute in if it was run sequentially. You can force the code inside the ORDERED construct to run in a predictable order while code outside of the construct runs in parallel.

The ORDERED and END ORDERED directives only take effect if you specify the -qsmp compiler option.

Syntax

```
ORDERED

block

END ORDERED
```

*block* represents the block of code that will be executed in sequence.

Rules

The ORDERED directive can only appear in the dynamic extent of a DO or PARALLEL DO directive. It is illegal to branch into or out of an ORDERED construct.

The ORDERED directive binds to the nearest dynamically enclosing DO or PARALLEL DO directive. You must specify the ORDERED clause on the DO or PARALLEL DO directive to which the ORDERED construct binds.
**ORDERED** constructs that bind to different **DO** directives are independent of each other.

Only one thread can execute an **ORDERED** construct at a time. Threads enter the **ORDERED** construct in the order of the loop iterations. A thread will enter the **ORDERED** construct if all of the previous iterations have either executed the construct or will never execute the construct.

Each iteration of a parallel loop with an **ORDERED** construct can only execute that **ORDERED** construct once. Each iteration of a parallel loop can execute at most one **ORDERED** directive. An **ORDERED** construct cannot appear within the dynamic extent of a **CRITICAL** construct.

The **END ORDERED** directive implies the **FLUSH** directive without the `variable_name_list`.

**Examples**

**Example 1:** In this example, an **ORDERED** parallel loop counts down.

```fortran
PROGRAM P
  !$OMP PARALLEL DO ORDERED
  DO I = 3, 1, -1
    !$OMP ORDERED
    PRINT *, I
  !$OMP END ORDERED
  END DO
END PROGRAM P
```

The expected output of this program is:

```
3
2
1
```

**Example 2:** This example shows a program with two **ORDERED** constructs in a parallel loop. Each iteration can only execute a single section.

```fortran
PROGRAM P
  !$OMP PARALLEL DO ORDERED
  DO I = 1, 3
    IF (MOD(I,2) == 0) THEN
      !$OMP ORDERED
      PRINT *, I*10
    !$OMP END ORDERED
    ELSE
      !$OMP ORDERED
      PRINT *, I
    !$OMP END ORDERED
    END IF
  END DO
END PROGRAM P
```

The expected output of this program is:

```
1
20
3
```

**Example 3:** In this example, the program computes the sum of all elements of an array that are greater than a threshold. **ORDERED** is used to ensure that the
results are always reproducible: roundoff will take place in the same order every
time the program is executed, so the program will always produce the same
results.

```fortran
PROGRAM P
    REAL :: A(1000)
    REAL :: THRESHOLD = 999.9
    REAL :: SUM = 0.0

    !$OMP PARALLEL DO ORDERED
    DO I = 1, 1000
        IF (A(I) > THRESHOLD) THEN
            !$OMP ORDERED
                SUM = SUM + A(I)
            !$OMP END ORDERED
        END IF
    END DO
    !$OMP END PARALLEL DO
END PROGRAM P
```

**Note:** To avoid bottleneck situations when using the ORDERED clause, you can try using DYNAMIC scheduling or STATIC scheduling with a small chunk size. See “SCHEDULE” on page 94 for more information.

**Related reference**

- See Loop parallelization in the Language Reference
- See -qsmp in the Compiler Reference
  - “PARALLEL DO / END PARALLEL DO” on page 86
  - “DO / END DO” on page 73
  - “CRITICAL / END CRITICAL” on page 71
  - “SCHEDULE” on page 94

**PARALLEL / END PARALLEL**

**Purpose**

The PARALLEL construct enables you to define a block of code that can be executed by a team of threads concurrently. The PARALLEL construct includes a PARALLEL directive that is followed by one or more blocks of code, and ends with an END PARALLEL directive.

The PARALLEL and END PARALLEL directives only take effect if you specify the -qsmp compiler option.
Syntax

where parallel_clause is:

copyin_clause
default_clause
firstprivate_clause
if_clause
(num_threads_clause)
private_clause
reduction_clause
shared_clause

.copyin_clause
  See — “COPYIN” on page 119
.default_clause
  See — “DEFAULT” on page 121
.if_clause
  See — “IF” on page 123
.firstprivate_clause
  See — “FIRSTPRIVATE” on page 124
.num_threads_clause
  See — “NUM_THREADS” on page 127
.private_clause
  See — “PRIVATE” on page 128.
.reduction_clause
  See — “REDUCTION” on page 129
.shared_clause
  See — “SHARED” on page 134

Rules

It is illegal to branch into or out of a PARALLEL construct.

The IF and DEFAULT clauses can appear at most once in a PARALLEL directive.
You should be careful when you perform input/output operations in a parallel region. If multiple threads execute a Fortran I/O statement on the same unit, you should make sure that the threads are synchronized. If you do not, the behavior is undefined. See “Parallel I/O issues” on page 253 for more information. Also note that although in the XL Fortran implementation each thread has exclusive access to the I/O unit, the OpenMP specification does not require exclusive access.

Directives that bind to a parallel region will bind to that parallel region even if it is serialized.

The END PARALLEL directive implies the FLUSH directive without the variable _name_list_.

**Examples**

**Example 1:** An example of an outer PARALLEL directive with the PRIVATE clause enclosing the PARALLEL construct. Note: The SHARED clause is present on the inner PARALLEL construct.

```
!$OMP PARALLEL PRIVATE(X)
!$OMP DO
  DO I = 1, 10
    X(I) = I
  END DO
!$OMP PARALLEL SHARED (X,Y)
!$OMP DO
  DO K = 1, 10
    Y(K,I) = K * X(I)
  END DO
!$OMP END DO
!$OMP END PARALLEL
```

**Example 2:** An example showing that a variable cannot appear in both a PRIVATE, and SHARED clause.

```
!$OMP PARALLEL PRIVATE(A), SHARED(A)
!$OMP DO
  DO I = 1, 1000
    A(I) = I * I
  END DO
!$OMP END DO
!$OMP END PARALLEL
```

**Example 3:** This example demonstrates the use of the COPYIN clause. Each thread created by the PARALLEL directive has its own copy of the common block _BLOCK_. The COPYIN clause causes the initial value of _FCTR_ to be copied into the threads that execute iterations of the DO loop.

```
PROGRAM TT
COMMON /BLOCK/ FCTR
INTEGER :: I, FCTR
!$OMP THREADPRIVATE(/BLOCK/)
      INTEGER :: A(100)
FCTR = -1
A = 0
!$OMP PARALLEL COPYIN(FCTR)
!$OMP DO
  DO I=1, 100
    FCTR = FCTR + I
    CALL SUB(A(I), I)
  END DO
!$OMP END PARALLEL
```

Chapter 7. Parallel programming with XL Fortran  85
ENDDO
!$OMP END PARALLEL

PRINT *, A
END PROGRAM

SUBROUTINE SUB(AA, J)
INTEGER :: FCTR, AA, J
COMMON /BLOCK/ FCTR
!$OMP THREADPRIVATE(/BLOCK/) ! EACH THREAD GETS ITS OWN COPY ! OF BLOCK.
AA = FCTR
FCTR = FCTR - J
END SUBROUTINE SUB

The expected output is:
0 1 2 3 ... 96 97 98 99

Related reference
“FLUSH” on page 77
“PARALLEL DO / END PARALLEL DO”
See INDEPENDENT in the Language Reference
“THREADPRIVATE” on page 108
“DO / END DO” on page 73
See -qdirective in the Compiler Reference
See -qsmp in the Compiler Reference

PARALLEL DO / END PARALLEL DO

Purpose

The PARALLEL DO directive enables you to specify which loops the compiler should parallelize. This is semantically equivalent to:

!$OMP PARALLEL
!$OMP DO
... 
!$OMP ENDDO
!$OMP END PARALLEL

and is a convenient way of parallelizing loops. The END PARALLEL DO directive allows you to indicate the end of a DO loop that is specified by the PARALLEL DO directive.

Type

The PARALLEL DO and END PARALLEL DO directives only take effect if you specify the -qsmp compiler option.
Syntax

where parallel_do_clause is:

collapse_clause
  copyin_clause
  default_clause
  firstprivate_clause
  IF (scalar_logical_expr)
  lastprivate_clause
  num_threads_clause
  ordered_clause
  private_clause
  reduction_clause
  SCHEDULE (sched_type, n)
  shared_clause

collapse_clause
  See — “COLLAPSE” on page 117

copyin_clause
  See — “COPYIN” on page 119

default_clause
  See — “DEFAULT” on page 121

if_clause
  See — “IF” on page 123.

firstprivate_clause
  See — “FIRSTPRIVATE” on page 124.

lastprivate_clause
  See — “LASTPRIVATE” on page 125.

num_threads_clause
  See — “NUM_THREADS” on page 127.

ordered_clause
  See — “ORDERED” on page 127

private_clause
  See — “PRIVATE” on page 128
Rules

The first noncomment line (not including other directives) that is following the PARALLEL DO directive must be a DO loop. This line cannot be an infinite DO or DO WHILE loop. The PARALLEL DO directive applies only to the DO loop that is immediately following the directive, and not to any nested DO loops, unless the COLLAPSE clause is specified.

If you specify a DO loop by a PARALLEL DO directive, the END PARALLEL DO directive is optional. If you use the END PARALLEL DO directive, it must immediately follow the end of the DO loop.

You may have a DO construct that contains several DO statements. If the DO statements share the same DO termination statement, and an END PARALLEL DO directive follows the construct, you can only specify a PARALLEL DO directive for the outermost DO statement of the construct.

You must not follow the PARALLEL DO directive by a DO (work-sharing) or DO SERIAL directive. You can specify only one PARALLEL DO directive for a given DO loop.

All work-sharing constructs and BARRIER directives that are encountered must be encountered in the same order by all threads in the team.

The PARALLEL DO directive must not appear with the INDEPENDENT directive for a given DO loop.

Note: You should use the PARALLEL DO directive for maximum portability across multiple vendors. The PARALLEL DO directive is a prescriptive directive, while the INDEPENDENT directive is an assertion about the characteristics of the loop. (See the INDEPENDENT directive in the XL Fortran Language Reference for more information.)

The IF clause may appear at most once in a PARALLEL DO directive.

An IF expression is evaluated outside of the context of the parallel construct. Any function reference in the IF expression must not have side effects.

By default, a nested parallel loop is serialized, regardless of the setting of the IF clause. You can change this default by using the -qsmp=nest_par compiler option.

If the REDUCTION variable of an inner DO loop appears in the PRIVATE or LASTPRIVATE clause of an enclosing DO loop or PARALLEL SECTIONS construct, the variable must be initialized before the inner DO loop.
A variable that appears in the REDUCTION clause of an INDEPENDENT directive of an enclosing DO loop must not also appear in the data_scope_entity_list of the PRIVATE or LASTPRIVATE clause.

Within a PARALLEL DO construct, variables that do not appear in the PRIVATE clause are assumed to be shared by default.

You should be careful when you perform input/output operations in a parallel region. If multiple threads execute a Fortran I/O statement on the same unit, you should make sure that the threads are synchronized. If you do not, the behavior is undefined. Also note that although in the XL Fortran implementation each thread has exclusive access to the I/O unit, the OpenMP specification does not require exclusive access.

Directives that bind to a parallel region will bind to that parallel region even if it is serialized.

The END PARALLEL DO directive implies the FLUSH directive without the variable_name_list.

Examples

Example 1: A valid example with the LASTPRIVATE clause.

```fortran
!$OMP PARALLEL DO PRIVATE(I), LASTPRIVATE(X)
DO I = 1,10
   X = I * I
   A(I) = X * B(I)
END DO
PRINT *, X          ! X has the value 100
```

Example 2: A valid example with the REDUCTION clause.

```fortran
!$OMP PARALLEL DO PRIVATE(I), REDUCTION(+:MYSUM)
DO I = 1,10
   MYSUM = MYSUM + IARR(I)
END DO
```

Example 3: A valid example where more than one thread accesses a variable that is marked as SHARED, but the variable is used only in a CRITICAL construct.

```fortran
!$OMP PARALLEL DO SHARED(X)
!$OMP PARALLEL DO PRIVATE(I)
DO I = 1,10
   A(I) = A(I) * I
   X = X + A(I)
!$OMP CRITICAL
   X = X + A(I)
!$OMP END CRITICAL
END DO
```

Example 4: A valid example of the END PARALLEL DO directive.

```fortran
REAL A(100), B(2:100), C(100)
!$OMP PARALLEL DO
   DO I = 2, 100
      B(I) = (A(I) + A(I-1))/2.0
   END DO
!$OMP END PARALLEL DO
!$OMP PARALLEL DO
   DO J = 1, 100
      C(J) = X + COS(J*5.5)
   END DO
!$OMP END PARALLEL DO
END
```
PARALLEL SECTIONS / END PARALLEL SECTIONS

Purpose

The PARALLEL SECTIONS construct enables you to define independent blocks of code that the compiler can execute concurrently. The PARALLEL SECTIONS construct includes a PARALLEL SECTIONS directive followed by one or more blocks of code delimited by the SECTION directive, and ends with an END PARALLEL SECTIONS directive.

The PARALLEL SECTIONS, SECTION and END PARALLEL SECTIONS directives only take effect if you specify the -qsmp compiler option.

Syntax

```
PARALLEL SECTIONS
   parallel_sections_clause
   block
   SECTION
   block
END PARALLEL SECTIONS
```

where `parallel_sections_clause` is:
Rules

The PARALLEL SECTIONS construct includes the delimiting directives, and the blocks of code they enclose. The rules below also refer to sections. You define a section as the block of code within the delimiting directives.

The SECTION directive marks the beginning of a block of code. At least one SECTION and its block of code must appear within the PARALLEL SECTIONS construct. Note, however, that you do not have to specify the SECTION directive for the first section. The end of a block is delimited by either another SECTION directive or by the END PARALLEL SECTIONS directive.

You can use the PARALLEL SECTIONS construct to specify parallel execution of the identified sections of code. There is no assumption as to the order in which sections are executed. Each section must not interfere with any other section in the construct unless the interference occurs within a CRITICAL construct.

It is illegal to branch into or out of any block of code that is defined by the PARALLEL SECTIONS construct.

The compiler determines how to divide the work among the threads based on a number of factors, such as the number of threads and the number of sections to be
executed in parallel. Therefore, a single thread may execute more than one
SECTION, or a thread may not execute any SECTION.

All work-sharing constructs and BARRIER directives that are encountered must be encountered in the same order by all threads in the team.

Within a PARALLEL SECTIONS construct, variables that are not appearing in the PRIVATE clause are assumed to be SHARED by default.

In a PARALLEL SECTIONS construct, a variable that appears in the REDUCTION clause of an INDEPENDENT directive or the PARALLEL DO directive of an enclosing DO loop must not also appear in the data_scope_entity_list of the PRIVATE clause.

If the REDUCTION variable of the inner PARALLEL SECTIONS construct appears in the PRIVATE clause of an enclosing DO loop or PARALLEL SECTIONS construct, the variable must be initialized before the inner PARALLEL SECTIONS construct.

The PARALLEL SECTIONS construct must not appear within a CRITICAL construct.

You should be careful when you perform input/output operations in a parallel region. If multiple threads execute a Fortran I/O statement on the same unit, you should make sure that the threads are synchronized. If you do not, the behavior is undefined. Also note that although in the XL Fortran implementation each thread has exclusive access to the I/O unit, the OpenMP specification does not require exclusive access.

Directives that bind to a parallel region will bind to that parallel region even if it is serialized.

The END PARALLEL SECTIONS directive implies the FLUSH directive without the variable_name_list.

**Examples**

**Example 1:**

```fortran
 !$OMP PARALLEL SECTIONS
 !$OMP SECTION
   DO I = 1, 10
     C(I) = MAX(A(I),A(I+1))
   END DO
 !$OMP SECTION
   W = U + V
   Z = X + Y
 !$OMP END PARALLEL SECTIONS
```

**Example 2:** In this example, the index variable I is declared as PRIVATE. Note also that the first optional SECTION directive has been omitted.

```fortran
 !$OMP PARALLEL SECTIONS PRIVATE(I)
   DO I = 1, 100
     A(I) = A(I) + 1
   END DO
 !$OMP SECTION
   CALL NORMALIZE (B)
   DO I = 1, 100
     B(I) = B(I) + 1.0
 ```
END DO
!$OMP SECTION
   DO I = 1, 100
      C(I) = C(I) * C(I)
   END DO
!$OMP END PARALLEL SECTIONS

Example 3: This example is invalid because there is a data dependency for the variable C across sections.

!$OMP PARALLEL SECTIONS
!$OMP SECTION
!$OMP SECTION
   DO I = 1, 10
      C(I) = C(I) * I
   END DO
!$OMP SECTION
   DO K = 1, 10
      D(K) = C(K) + K
   END DO
!$OMP END PARALLEL SECTIONS

Related reference

"PARALLEL / END PARALLEL" on page 83
"PARALLEL DO / END PARALLEL DO" on page 86
See INDEPENDENT in the Language Reference
"THREADPRIVATE" on page 108
See -qdirective in the Compiler Reference
See -qsmp in the Compiler Reference

PARALLEL WORKSHARE / END PARALLEL WORKSHARE

Purpose

The PARALLEL WORKSHARE construct provides a short form method for including a WORKSHARE directive inside a PARALLEL construct.

The PARALLEL WORKSHARE / END PARALLEL WORKSHARE directives only take effect if you specify the -qsmp compiler option

Syntax

```plaintext
PARALLEL WORKSHARE
   parallel_workshare_clause
END PARALLEL WORKSHARE
```
where `parallel_workshare_clause` is any of the directives accepted by either the `PARALLEL` or `WORKSHARE` directives.

Related reference

- “PARALLEL / END PARALLEL” on page 83
- “WORKSHARE” on page 113

**SCHEDULE**

**Purpose**

The `SCHEDULE` directive allows the user to specify the chunking method for parallelization. Work is assigned to threads in different manners depending on the scheduling type or chunk size used.

The `SCHEDULE` directive only takes effect if you specify the `-qsmp` compiler option.

**Syntax**

```fortran
SCHEDULE(("sched_type", n))
```

- `n` must be a positive, specification expression. You must not specify `n` for the `sched_type` `RUNTIME`.
- `sched_type` is `AFFINITY`, `DYNAMIC`, `GUIDED`, `RUNTIME`, or `STATIC`.

For more information on `sched_type` parameters, see the `SCHEDULE` clause.

- `number_of_iterations` is the number of iterations in the loop to be parallelized.
- `number_of_threads` is the number of threads used by the program.

**Rules**

The `SCHEDULE` directive must appear in the specification part of a scoping unit.

Only one `SCHEDULE` directive may appear in the specification part of a scoping unit.

The `SCHEDULE` directive applies to one of the following:

- All loops in the scoping unit that do not already have explicit scheduling types specified. Individual loops can have scheduling types specified using the `SCHEDULE` clause of the `PARALLEL DO` directive.
- Loops that the compiler generates and have been chosen to be parallelized by automatic parallelization. For example, the `SCHEDULE` directive applies to loops generated for `FORALL`, `WHERE`, I/O implied-DO, and array constructor implied-DO.

Any dummy arguments appearing or referenced in the specification expression for the chunk size `n` must also appear in the `SUBROUTINE` or `FUNCTION` statement and in all `ENTRY` statements appearing in the given subprogram.
If the specified chunk size \( n \) is greater than the number of iterations, the loop will not be parallelized and will execute on a single thread.

If you specify more than one method of determining the chunking algorithm, the compiler will follow, in order of precedence:

1. **SCHEDULE** clause to the **PARALLEL DO** directive.
2. **SCHEDULE** directive
3. schedule suboption to the `-qsmp` compiler option. See the `-qsmp` option in the XL Fortran Compiler Reference
4. **XLSMPOPTS** runtime option. See “The XLSMPOPTS environment variable” on page 58
5. runtime default (that is, **STATIC**)

**Examples**

**Example 1.** Given the following information:

\[
\begin{align*}
\text{number of iterations} & = 1000 \\
\text{number of threads} & = 4
\end{align*}
\]

and using the **GUIDED** scheduling type, the chunk sizes would be as follows:

\[
250 \ 188 \ 141 \ 106 \ 79 \ 59 \ 45 \ 33 \ 25 \ 19 \ 14 \ 11 \ 8 \ 6 \ 4 \ 3 \ 3 \ 2 \ 1 \ 1 \ 1 \ 1
\]

The iterations would then be divided into the following chunks:

\[
\begin{align*}
\text{chunk 1} & = \text{iterations 1 to 250} \\
\text{chunk 2} & = \text{iterations 251 to 438} \\
\text{chunk 3} & = \text{iterations 439 to 579} \\
\text{chunk 4} & = \text{iterations 580 to 685} \\
\text{chunk 5} & = \text{iterations 686 to 764} \\
\text{chunk 6} & = \text{iterations 765 to 823} \\
\text{chunk 7} & = \text{iterations 824 to 868} \\
\text{chunk 8} & = \text{iterations 869 to 901} \\
\text{chunk 9} & = \text{iterations 902 to 926} \\
\text{chunk 10} & = \text{iterations 927 to 945} \\
\text{chunk 11} & = \text{iterations 946 to 959} \\
\text{chunk 12} & = \text{iterations 960 to 970} \\
\text{chunk 13} & = \text{iterations 971 to 978} \\
\text{chunk 14} & = \text{iterations 979 to 984} \\
\text{chunk 15} & = \text{iterations 985 to 988} \\
\text{chunk 16} & = \text{iterations 989 to 991} \\
\text{chunk 17} & = \text{iterations 992 to 994} \\
\text{chunk 18} & = \text{iterations 995 to 996} \\
\text{chunk 19} & = \text{iterations 997 to 997} \\
\text{chunk 20} & = \text{iterations 998 to 998} \\
\text{chunk 21} & = \text{iterations 999 to 999} \\
\text{chunk 22} & = \text{iterations 1000 to 1000}
\end{align*}
\]

A possible scenario for the division of work could be:

\[
\begin{align*}
\text{thread 1} & \ \text{executes chunks 1 5 10 13 18 20} \\
\text{thread 2} & \ \text{executes chunks 2 7 9 14 16 22} \\
\text{thread 3} & \ \text{executes chunks 3 6 12 15 19} \\
\text{thread 4} & \ \text{executes chunks 4 8 11 17 21}
\end{align*}
\]

**Example 2.** Given the following information:

\[
\begin{align*}
\text{number of iterations} & = 100 \\
\text{number of threads} & = 4
\end{align*}
\]

and using the **AFFINITY** scheduling type, the iterations would be divided into the following partitions:
The partitions would be divided into the following chunks:

chunk 1a = iterations 1 to 13
chunk 1b = iterations 14 to 19
chunk 1c = iterations 20 to 22
chunk 1d = iterations 23 to 24
chunk 1e = iterations 25 to 25

chunk 2a = iterations 26 to 38
chunk 2b = iterations 39 to 44
chunk 2c = iterations 45 to 47
chunk 2d = iterations 48 to 49
chunk 2e = iterations 50 to 50

chunk 3a = iterations 51 to 63
chunk 3b = iterations 64 to 69
chunk 3c = iterations 70 to 72
chunk 3d = iterations 73 to 74
chunk 3e = iterations 75 to 75

chunk 4a = iterations 76 to 88
chunk 4b = iterations 89 to 94
chunk 4c = iterations 95 to 97
chunk 4d = iterations 98 to 99
chunk 4e = iterations 100 to 100

A possible scenario for the division of work could be:

thread 1 executes chunks 1a 1b 1c 1d 1e 4d
thread 2 executes chunks 2a 2b 2c 2d
thread 3 executes chunks 3a 3b 3c 3d 3e 2e
thread 4 executes chunks 4a 4b 4c 4e

In this scenario, thread 1 finished executing all the chunks in its partition and then grabbed an available chunk from the partition of thread 4. Similarly, thread 3 finished executing all the chunks in its partition and then grabbed an available chunk from the partition of thread 2.

Example 3. Given the following information:

number of iterations = 1000
number of threads = 4

and using the DYNAMIC scheduling type and chunk size of 100, the chunk sizes would be as follows:

100 100 100 100 100 100 100 100 100 100

The iterations would be divided into the following chunks:

chunk 1 = iterations 1 to 100
chunk 2 = iterations 101 to 200
chunk 3 = iterations 201 to 300
chunk 4 = iterations 301 to 400
chunk 5 = iterations 401 to 500
chunk 6 = iterations 501 to 600
chunk 7 = iterations 601 to 700
chunk 8 = iterations 701 to 800
chunk 9 = iterations 801 to 900
chunk 10 = iterations 901 to 1000

A possible scenario for the division of work could be:
Example 4. Given the following information:

number of iterations = 100
number of threads = 4

and using the STATIC scheduling type, the iterations would be divided into the following chunks:

chunk 1 = iterations 1 to 25
chunk 2 = iterations 26 to 50
chunk 3 = iterations 51 to 75
chunk 4 = iterations 76 to 100

A possible scenario for the division of work could be:

thread 1 executes chunks 1
thread 2 executes chunks 2
thread 3 executes chunks 3
thread 4 executes chunks 4

Related reference

See DO in the Language Reference

SECTIONS / END SECTIONS

Purpose

The SECTIONS construct defines distinct blocks of code to be executed in parallel by threads in the team.

The SECTIONS and END SECTIONS directives only take effect if you specify the -qsmp compiler option.

Syntax

where sections_clause is:
**Rules**

The **SECTIONS** construct must be encountered by all threads in a team or by none of the threads in a team. All work-sharing constructs and **BARRIER** directives that are encountered must be encountered in the same order by all threads in the team.

The **SECTIONS** construct includes the delimiting directives, and the blocks of code they enclose. At least one block of code must appear in the construct.

You must specify the **SECTION** directive at the beginning of each block of code except for the first. The end of a block is delimited by either another **SECTION** directive or by the **END SECTIONS** directive.

It is illegal to branch into or out of any block of code that is enclosed in the **SECTIONS** construct. All **SECTION** directives must appear within the lexical extent of the **SECTIONS/END SECTIONS** directive pair.

The scheduling of structured blocks among threads in the team is set so that the first thread arriving is the first thread to execute the block. The compiler determines how to divide the work among the threads based on a number of factors, such as the number of threads in the team and the number of sections to be executed in parallel. In a **SECTIONS** construct, a single thread might execute more than one **SECTION**. It is also possible that a thread in the team might not execute any **SECTION**.

In order for the directive to execute in parallel, you must place the **SECTIONS/END SECTIONS** pair within the dynamic extent of a parallel region. Otherwise, the blocks will be executed serially.

If you specify **NOWAIT** on the **SECTIONS** directive, a thread that completes its sections early will proceed to the instructions following the **SECTIONS** construct. If you do not specify the **NOWAIT** clause, each thread will wait for all of the other threads in the same team to reach the **END SECTIONS** directive. However, there is no implied **BARRIER** at the start of the **SECTIONS** construct.

You cannot specify a **SECTIONS** directive within the dynamic extent of a **CRITICAL, MASTER, ORDERED, or TASK** directive.

You cannot nest **SECTIONS, DO or SINGLE** directives that bind to the same **PARALLEL** directive.
**BARRIER** and **MASTER** directives are not permitted in the dynamic extent of a **SECTIONS** directive.

The **END SECTIONS** directive implies the **FLUSH** directive.

### Examples

**Example 1:** This example shows a valid use of the **SECTIONS** construct within a **PARALLEL** region.

```fortran
INTEGER :: I, B(500), S, SUM
!
S = 0
SUM = 0
!$OMP PARALLEL SHARED(SUM), FIRSTPRIVATE(S)
!$OMP SECTIONS REDUCTION(+: SUM), LASTPRIVATE(I)
!$OMP SECTION
   S = FCT1(B(1::2)) ! Array B is not altered in FCT1.
   SUM = SUM + S
! ...
!$OMP SECTION
   S = FCT2(B(2::2)) ! Array B is not altered in FCT2.
   SUM = SUM + S
! ...
!$OMP SECTION
   DO I = 1, 500      ! The local copy of S is initialized
      S = S + B(I)    ! to zero.
   END DO
   SUM = SUM + S
! ...
!$OMP END SECTIONS
! ...
!$OMP DO REDUCTION(-: SUM)
   DO J=I-1, 1, -1   ! The loop starts at 500 -- the last
      SUM = SUM - B(J) ! value from the previous loop.
   END DO
!$OMP MASTER
   SUM = SUM - FCT1(B(1::2)) - FCT2(B(2::2))
!$OMP END MASTER
!$OMP END PARALLEL
!
! Upon termination of the PARALLEL
! region, the value of SUM remains zero.
```

**Example 2:** This example shows a valid use of nested **SECTIONS**.

```fortran
!$OMP PARALLEL
!$OMP MASTER
   CALL RANDOM_NUMBER(CX)
   CALL RANDOM_NUMBER(CY)
   CALL RANDOM_NUMBER(CZ)
!$OMP END MASTER
!$OMP SECTIONS
!$OMP SECTION
!$OMP PARALLEL
!$OMP SECTIONS PRIVATE(I)
!$OMP SECTION
   DO I=1, 5000
      X(I) = X(I) + CX
   END DO
!$OMP SECTION
   DO I=1, 5000
      Y(I) = Y(I) + CY
   END DO
```
END DO
!$OMP END SECTIONS
!$OMP END PARALLEL

!$OMP SECTION
!$OMP PARALLEL SHARED(CZ,Z)
!$OMP DO
DO I=1, 5000
   Z(I) = Z(I) + CZ
END DO
!$OMP END DO
!$OMP END PARALLEL
!$OMP END SECTIONS NOWAIT

! The following computations do not
! depend on the results from the
! previous section.

!$OMP DO
DO I=1, 5000
   T(I) = T(I) * CT
END DO
!$OMP END DO
!$OMP END PARALLEL

Related reference
“PARALLEL / END PARALLEL” on page 83
“BARRIER” on page 70
“PARALLEL DO / END PARALLEL DO” on page 86
See INDEPENDENT in the Language Reference
“THREADPRIVATE” on page 108
See -qdirective in the Compiler Reference
See -qsmp in the Compiler Reference

SINGLE / END SINGLE

Purpose
You can use the SINGLE / END SINGLE directive construct to specify that the
enclosed code should only be executed by one thread in the team.

The SINGLE directive only takes effect if you specify the -qsmp compiler option.

Syntax
where *single_clause* is:

```
  private_clause
    firstprivate_clause
```

*private_clause*

See — “PRIVATE” on page 128.

*firstprivate_clause*

See — “FIRSTPRIVATE” on page 124.

where *end_single_clause* is:

```
  copyprivate_clause
```

**NOWAIT**

*copyprivate_clause*

Rules

It is illegal to branch into or out of a block that is enclosed within the **SINGLE** construct.

The **SINGLE** construct must be encountered by all threads in a team or by none of the threads in a team. The first thread to encounter the **SINGLE** construct will execute it. All work-sharing constructs and **BARRIER** directives that are encountered must be encountered in the same order by all threads in the team.

If you specify **NOWAIT** on the **END SINGLE** directive, the threads that are not executing the **SINGLE** construct will proceed to the instructions following the **SINGLE** construct. If you do not specify the **NOWAIT** clause, each thread will wait at the **END SINGLE** directive until the thread executing the construct reaches the **END SINGLE** directive. You may not specify **NOWAIT** and **COPYPRIVATE** as part of the same **END SINGLE** directive.
There is no implied BARRIER at the start of the SINGLE construct. If you do not specify the NOWAIT clause, the BARRIER directive is implied at the END SINGLE directive.

You cannot nest SECTIONS, DO and SINGLE directives inside one another if they bind to the same PARALLEL directive.

SINGLE directives are not permitted within the dynamic extent of CRITICAL, MASTER, ORDERED, or TASK directives. BARRIER and MASTER directives are not permitted within the dynamic extent of SINGLE directives.

If you have specified a variable as PRIVATE, FIRSTPRIVATE, LASTPRIVATE or REDUCTION in the PARALLEL construct which encloses your SINGLE construct, you cannot specify the same variable in the PRIVATE or FIRSTPRIVATE clause of the SINGLE construct.

The SINGLE directive binds to the closest dynamically enclosing PARALLEL directive, if one exists.

Examples

Example 1: In this example, the BARRIER directive is used to ensure that all threads finish their work before entering the SINGLE construct.

```
REAL :: X(100), Y(50)
! ...
!$OMP PARALLEL DEFAULT(SHARED)
CALL WORK(X)
!$OMP BARRIER
!$OMP SINGLE
  CALL OUTPUT(X)
  CALL INPUT(Y)
!$OMP END SINGLE
CALL WORK(Y)
!$OMP END PARALLEL
```

Example 2: In this example, the SINGLE construct ensures that only one thread is executing a block of code. In this case, array B is initialized in the DO (work-sharing) construct. After the initialization, a single thread is employed to perform the summation.

```
INTEGER :: I, J
REAL :: B(500,500), SM
! ...
J = ...
SM = 0.0
!$OMP PARALLEL
!$OMP DO PRIVATE(I)
!$OMP DO I=1, 500
  CALL INITARR(B(:,I)) ! initialize the array B
ENDDO
!$OMP END DO
!$OMP SINGLE ! employ only one thread
  DO I=1, 500
    SM = SM + SUM(B(:,I))
  ENDDO
!$OMP END SINGLE
!$OMP DO PRIVATE(I)
```
DO I=500, 1, -1  
   CALL INITARR(B(I,:), 501-I)  ! re-initialize the array B  
ENDDO  
!$OMP END PARALLEL

**Example 3:** This example shows a valid use of the PRIVATE clause. Array $X$ is PRIVATE to the SINGLE construct. If you were to reference array $X$ immediately following the construct, it would be undefined.

REAL :: $X(2000)$, $A(1000)$, $B(1000)$

!$OMP PARALLEL
!$OMP SINGLE PRIVATE($X$)
   CALL READ_IN_DATA($X$)
   $A$ = $X(1::2)$
   $B$ = $X(2::2)$
!$OMP END SINGLE
!$OMP END PARALLEL

**Example 4:** In this example, the LASTPRIVATE variable $I$ is used in allocating $TMP$, the PRIVATE variable in the SINGLE construct.

SUBROUTINE ADD($A$, $UPPERBOUND$)  
   INTEGER :: $A(UPPERBOUND)$, $I$, $UPPERBOUND$
   INTEGER, ALLOCATABLE :: $TMP(:)$
   ! ...  
   !$OMP PARALLEL
   !$OMP DO LASTPRIVATE($I$)
      DO $I$=1, $UPPERBOUND$
         $A(I)$ = $I$ + 1
      ENDDO
   !$OMP END DO
   !$OMP SINGLE FIRSTPRIVATE($I$), PRIVATE($TMP$)
   ALLOCATE($TMP(0:I-1)$)
   $TMP$ = (/ $A(J), J=I,1,-1$ /)
   ! ...
   DEALLOCATE($TMP$)
   !$OMP END SINGLE
   !$OMP END PARALLEL
   ! ...
END SUBROUTINE ADD

**Example 5:** In this example, a value for the variable $I$ is entered by the user. This value is then copied into the corresponding variable $I$ for all other threads in the team using a COPYPRIVATE clause on an END SINGLE directive.

INTEGER $I$
   !$OMP PARALLEL PRIVATE ($I$)
   ! ...  
   !$OMP SINGLE
   READ (*, *) $I$
   !$OMP END SINGLE COPYPRIVATE ($I$)  ! In all threads in the team, $I$
   ! is equal to the value
   ! that you entered.
   ! ...
   !$OMP END PARALLEL

**Example 6:** In this example, variable $J$ with a POINTER attribute is specified in a COPYPRIVATE clause on an END SINGLE directive. The value of $J$, not the value of the object that it points to, is copied into the corresponding variable $J$ for all other threads in the team. The object itself is shared among all the threads in the team.
INTEGER, POINTER :: J

!$OMP PARALLEL PRIVATE (J)
! ... !$OMP SINGLE
ALLOCATE (J)
READ (*, *) J !$OMP END SINGLE COPYPRIVATE (J)
!$OMP ATOMIC
J = J + OMP_GET_THREAD_NUM()
!$OMP BARRIER
!$OMP SINGLE
WRITE (*, *) 'J = ', J ! The result is the sum of all values added to ! J. This result shows that the pointer object ! is shared by all threads in the team.

DEALLOCATE (J)
!$OMP END SINGLE
!$OMP END PARALLEL

Related reference
“BARRIER” on page 70
“CRITICAL / END CRITICAL” on page 71
“FLUSH” on page 77
“MASTER / END MASTER” on page 80
“PARALLEL / END PARALLEL” on page 83

TASK

Purpose

The OpenMP TASK directive can be used to explicitly define a task.

Use the TASK directive when you want to identify a block of code to be executed in parallel with the code outside the task region. The TASK directive can be useful for parallelizing irregular algorithms such as pointer chasing or recursive algorithms for which other OpenMP workshare constructs are inadequate. The TASK directive only takes effect if you specify the -qsmp compiler option.

Syntax

where task_clause is:
**Rules**

If no **DEFAULT** clause is present with the enclosing **TASK** construct that is not lexically enclosed by a parallel region, dummy arguments that do not appear in any **PRIVATE**, **FIRSTPRIVATE**, **LASTPRIVATE**, or **SHARED** clause of the enclosing **TASK** construct are firstprivate.

If no **DEFAULT** clause is present with the enclosing **TASK** construct, a variable that is private in the innermost enclosing parallel construct is firstprivate in the **TASK** construct.

If no **DEFAULT** clause is present with the enclosing **TASK** construct, local variables of a routine are firstprivate if there is no enclosing parallel construct.

**Related reference**

"TASKWAIT"

"UNTIED" on page 136

**TASKWAIT**

**Purpose**

Use the **TASKWAIT** directive to specify a *wait* for child tasks to be completed that are generated by the current task.

**Syntax**

```fortran
>>- TASKWAIT
```
THREA DLOCAL

Purpose

You can use the THREADLOCAL directive to declare thread-specific common data. It is a possible method of ensuring that access to data that is contained within COMMON blocks is serialized.

In order to make use of this directive it is not necessary to specify the -qsmp compiler option, but the invocation command must be xlf_r, xlf_r7, xlf90_r, xlf90_r7, xlf95_r, xlf95_r7, or xlf2003_r to link the necessary libraries.

Syntax

```
 THREADLOCAL / common_block_name /
```

Rules

You can only declare named blocks as THREADLOCAL. All rules and constraints that normally apply to named common blocks apply to common blocks that are declared as THREADLOCAL. See the COMMON statement in the XL Fortran Language Reference for more information on the rules and constraints that apply to named common blocks.

The THREADLOCAL directive must appear in the specification_part of the scoping unit. If a common block appears in a THREADLOCAL directive, it must also be declared within a COMMON statement in the same scoping unit. The THREADLOCAL directive may occur before or after the COMMON statement. See [Main program] in the XL Fortran Language Reference for more information on the specification_part of the scoping unit.

A common block cannot be given the THREADLOCAL attribute if it is declared within a PURE subprogram.

Members of a THREADLOCAL common block must not appear in NAMELIST statements.

A common block that is use-associated must not be declared as THREADLOCAL in the scoping unit that contains the USE statement.

Any pointers declared in a THREADLOCAL common block are not affected by the -qinit=f90ptr compiler option.

Objects within THREADLOCAL common blocks may be used in parallel loops and parallel sections. However, these objects are implicitly shared across the iterations of the loop, and across code blocks within parallel sections. In other
words, within a scoping unit, all accessible common blocks, whether declared as 
THREADLOCAL or not, have the SHARED attribute within parallel loops and 
sections in that scoping unit.

If a common block is declared as THREADLOCAL within a scoping unit, any 
subprogram that declares or references the common block, and that is directly or 
indirectly referenced by the scoping unit, must be executed by the same thread 
executing the scoping unit. If two procedures that declare common blocks are 
executed by different threads, then they would obtain different copies of the 
common block, provided that the common block had been declared 
THREADLOCAL. Threads can be created in one of the following ways:

• Explicitly, via pthreads library calls
• Implicitly by the compiler for parallel loop execution
• Implicitly by the compiler for parallel section execution.

If a common block is declared to be THREADLOCAL in one scoping unit, it must 
be declared to be THREADLOCAL in every scoping unit that declares the 
common block.

If a THREADLOCAL common block that does not have the SAVE attribute is 
declared within a subprogram, the members of the block become undefined at 
subprogram RETURN or END, unless there is at least one other scoping unit in 
which the common block is accessible that is making a direct or indirect reference 
to the subprogram.

You cannot specify the same common_block_name for both a THREADLOCAL 
directive and a THREADPRIVATE directive.

Example 1: The following procedure "FORT_SUB" is invoked by two threads:

SUBROUTINE FORT_SUB(IARG)
   INTEGER IARG
   
   CALL LIBRARY_ROUTINE1()
   CALL LIBRARY_ROUTINE2()
   ...
END SUBROUTINE FORT_SUB

SUBROUTINE LIBRARY_ROUTINE1()
   COMMON /BLOCK/ R
   SAVE /BLOCK/
   IBM* THREADLOCAL /BLOCK/
   R = 1.0
   ...
END SUBROUTINE LIBRARY_ROUTINE1

SUBROUTINE LIBRARY_ROUTINE2()
   COMMON /BLOCK/ R
   SAVE /BLOCK/
   IBM* THREADLOCAL /BLOCK/
   ...
   = R
   ...
END SUBROUTINE LIBRARY_ROUTINE2

Example 2: "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
    INTEGER AA
    COMMON /BLOCK/ J
    INTEGER :: J
    !IBM* THREADLOCAL /BLOCK/
    ...
    AA = J
    ! The value of 'J' is undefined.
END SUBROUTINE ANOTHER_SUB

One or more sample programs under the directory /usr/lpp/xlf/samples/modules/threadlocal illustrate how to use the THREADLOCAL directive and create threads in C.

Related reference
- See -qdirective in the Compiler Reference
- See -qinit in the Compiler Reference
- See COMMON in the Language Reference
- See Main program in the Language Reference

THREADPRIVATE
Purpose

The THREADPRIVATE directive allows you to specify named common blocks and named variables as private to a thread but global within that thread. Once you declare a common block or variable THREADPRIVATE, each thread in the team maintains a separate copy of that common block or variable. Data written to a THREADPRIVATE common block or variable remains private to that thread and is not visible to other threads in the team.

In the serial and MASTER sections of a program, only the master thread’s copy of the named common block and variable is accessible.

Use the COPYIN clause on the PARALLEL, PARALLEL DO, PARALLEL SECTIONS or PARALLEL WORKSHARE directives to specify that upon entry into a parallel region, data in the master thread’s copy of a named common block or named variable is copied to each thread’s private copy of that common block or variable.

The THREADPRIVATE directive only takes effect if you specify the -qsmp compiler option.
Syntax

```fortran
THREADPRIVATE(—threadprivate_entity_list—)
```

where `threadprivate_entity_list` is:

```fortran
variable_name

/ common_block_name /
```

`common_block_name` is the name of a common block to be made private to a thread.

`variable_name` is the name of a variable to be made private to a thread.

**Rules**

You cannot specify a `THREADPRIVATE` variable, common block, or the variables that comprise that common block in a `PRIVATE`, `FIRSTPRIVATE`, `LASTPRIVATE`, `SHARED`, or `REDUCTION` clause.

A `THREADPRIVATE` variable must have the `SAVE` attribute. For variables or common blocks declared in the scope of a module, the `SAVE` attribute is implied. If you declare the variable outside of the scope of the module, the `SAVE` attribute must be specified.

In `THREADPRIVATE` directives, you can only specify named variables and named common blocks.

A variable can only appear in a `THREADPRIVATE` directive in the scope in which it is declared, and a `THREADPRIVATE` variable or common block may only appear once in a given scope. The variable must not be an element of a common block, or be declared in an `EQUIVALENCE` statement.

You cannot specify the same `common_block_name` for both a `THREADPRIVATE` directive and a `THREADLOCAL` directive.

All rules and constraints that apply to named common blocks also apply to common blocks declared as `THREADPRIVATE`. See the `COMMON statement` in the *XL Fortran Language Reference*.

If you declare a common block as `THREADPRIVATE` in one scoping unit, you must declare it as `THREADPRIVATE` in all other scoping units in which it is declared.

On entry into any parallel region, a `THREADPRIVATE` variable, or a variable in a `THREADPRIVATE` common block specified in a `COPYIN` clause is subject to the criteria stated in the `Rules` section for the `COPYIN` clause.

On entry into the first parallel region of the program, `THREADPRIVATE` variables or variables within a `THREADPRIVATE` common block not specified in a `COPYIN` clause are subject to the following criteria:
• If the variable has the **ALLOCATABLE** attribute, the initial allocation status of each copy of that variable is not currently allocated.

• If the variable has the **POINTER** attribute, and that pointer is disassociated through either explicit or default initialization, the association status of each copy of that variable is disassociated. Otherwise, the association status of the pointer is undefined.

• If the variable has neither the **ALLOCATABLE** nor the **POINTER** attribute and is defined through either explicit or default initialization, then each copy of that variable is defined. If the variable is undefined, then each copy of that variable is undefined.

On entry into subsequent parallel regions of the program, **THREADPRIVATE**

variables, or variables within a **THREADPRIVATE** common block not specified in a **COPYIN** clause, are subject to the following criteria:

• If you are using the **OMP_DYNAMIC** environment variable, or the **omp_set_dynamic** subroutine to enable dynamic threads and:
  - If the number of threads is smaller than the number of threads in the previous region, and if a **THREADPRIVATE** object is referenced in both regions, then threads with the same thread number in their respective regions will reference the same copy of that variable.
  - If the number of threads is larger than the number of threads in the previous region, then the definition and association status of a **THREADPRIVATE**
    object is undefined, and the allocation status is undefined.

• If dynamic threads are disabled, the definition, association, or allocation status and definition, if the thread’s copy of the variable was defined, is retained.

You cannot access the name of a common block by use association or host association. Thus, a named common block can only appear on a **THREADPRIVATE**

directive if the common block is declared in the scoping unit that contains the **THREADPRIVATE** directive. However, you can access the variables in the common block by use association or host association. For more information, see **Host** and **Use association** in the *XL Fortran Language Reference*.

The **-qinit=f90ptr** compiler option does not affect pointers that you have declared in a **THREADPRIVATE** common block.

The **DEFAULT** clause does not affect variables in **THREADPRIVATE** common blocks.

**Examples**

**Example 1**: In this example, the **PARALLEL DO** directive invokes multiple threads that call **SUB1**. The common block **BLK** in **SUB1** shares the data that is specific to the thread with subroutine **SUB2**, which is called by **SUB1**.

```fortran
PROGRAM TT
  INTEGER :: I, B(50)
!$OMP PARALLEL DO SCHEDULE(STATIC, 10)
  DO I=1, 50
      CALL SUB1(I, B(I)) ! Multiple threads call SUB1.
  ENDDO
END PROGRAM TT

SUBROUTINE SUB1(J, X)
  INTEGER :: J, X, A(100)
  COMMON /BLK/ A
```
Array \( a \) is private to each thread.

```fortran
CALL SUB2(J)
X = A(J) + A(J + 50)
END SUBROUTINE SUB1

SUBROUTINE SUB2(K)
INTEGER :: C(100)
COMMON /BLK/ C
!$OMP THREADPRIVATE(/BLK/)
! ...
C = K
! ...
END SUBROUTINE SUB2
```

Since each thread has its own copy of the common block BLK, the assignment of array C has no effect on the copies of that block owned by other threads.

Example 2: In this example, each thread has its own copy of the common block ARR in the parallel section. If one thread initializes the common block variable TEMP, the initial value is not visible to other threads.

```fortran
PROGRAM ABC
INTEGER :: I, TEMP(100), ARR1(50), ARR2(50)
COMMON /ARR/ TEMP
!$OMP THREADPRIVATE(/ARR/)
INTERFACE
SUBROUTINE SUBS(X)
INTEGER :: X(:)
END SUBROUTINE
END INTERFACE

! ...
!$OMP PARALLEL SECTIONS
!$OMP SECTION
! ...
TEMP(1:100:2) = -1
TEMP(2:100:2) = 2
CALL SUBS(ARR1)
! ...
!$OMP SECTION
! ...
TEMP(1:100:2) = 1
TEMP(2:100:2) = -2
CALL SUBS(ARR2)
! ...
!$OMP END PARALLEL SECTIONS
! ...
PRINT *, SUM(ARR1), SUM(ARR2)
END PROGRAM
```

The expected output for this program is:

50 -50
Example 3: In the following example, local variables outside of a common block are declared THREADPRIVATE.

```fortran
MODULE MDL
  INTEGER :: A(2)
  INTEGER, POINTER :: P
  INTEGER, TARGET :: T
  !$OMP THREADPRIVATE(A, P)
END MODULE MDL

PROGRAM MVAR
  USE MDL
  INTEGER :: I
  INTEGER OMP_GET_THREAD_NUM
  CALL OMP_SET_NUM_THREADS(2)
  A = (/1, 2/)  
  T = 4  
  P => T
  !$OMP PARALLEL PRIVATE(I) COPYIN(A, P)
  I = OMP_GET_THREAD_NUM()
  IF (I .EQ. 0) THEN
    A(1) = 100
    T = 5
  ELSE IF (I .EQ. 1) THEN
    A(2) = 200
  END IF
  !$OMP END PARALLEL

  !$OMP PARALLEL PRIVATE(I)
  I = OMP_GET_THREAD_NUM()
  IF (I .EQ. 0) THEN
    PRINT *, 'A(2)', A(2)
  ELSE IF (I .EQ. 1) THEN
    PRINT *, 'A(1)', A(1)
    PRINT *, 'P => ', P
  END IF
  !$OMP END PARALLEL

END PROGRAM MVAR
```

If dynamic threads mechanism is disabled, the expected output is:

A(2) = 2  
A(1) = 1  
P => 5

or

A(1) = 1  
P => 5  
A(2) = 2

Related reference

See COMMON in the Language Reference

“OMP_DYNAMIC environment variable” on page 63

“omp_set_dynamic(enable_expr)” on page 147

“PARALLEL / END PARALLEL” on page 83

“PARALLEL DO / END PARALLEL DO” on page 86

“PARALLEL SECTIONS / END PARALLEL SECTIONS” on page 90
WORKSHARE

Purpose

The WORKSHARE directive allows you to parallelize the execution of array operations. A WORKSHARE directive divides the tasks associated with an enclosed block of code into units of work. When a team of threads encounters a WORKSHARE directive, the threads in the team share the tasks, so that each unit of work executes exactly once.

The WORKSHARE directive only takes effect if you specify the -qsmp compiler option.

Syntax

```
WORKSHARE

block

END WORKSHARE
```

`block` is a structured block of statements that allows work sharing within the lexical extent of the WORKSHARE construct. The execution of statements are synchronized so that statements whose result is a dependent on another statement are evaluated before that result is required. The `block` can contain any of the following:

- Array assignment statements
- ATOMIC directives
- CRITICAL constructs
- FORALL constructs
- FORALL statements
- PARALLEL construct
- PARALLEL DO construct
- PARALLEL SECTION construct
- PARALLEL WORKSHARE construct
- Scalar assignment statements
- WHERE constructs
- WHERE statements

The transformational intrinsic functions you can use as part of an array operation are:

- ALL
- ANY
- COUNT
- CSHIFT
- DOT_PRODUCT
- EOSHIFT
- MATMUL
- MAXLOC
- MAXVAL
- MINLOC
- MINVAL
- PACK
- PRODUCT
- RESHAPE
- SPREAD
- SUM
- TRANSPOSE
- UNPACK
The *block* can also contain statements bound to lexically enclosed
PARALLEL constructs. These statements are not restricted.

Any user-defined function calls within the block must be elemental.

Statements enclosed in a WORKSHARE directive are divided into units of work.
The definition of a unit of work varies according to the statement evaluated. A unit
of work is defined as follows:

- **Array expressions**: Evaluation of each element of an array expression is a unit of
  work. Any of the transformational intrinsic functions listed above may be
divided into any number of units of work.
- **Assignment statements**: In an array assignment statement, the assignment of
each element in the array is a unit of work. For scalar assignment statements, the
assignment operation is a unit of work.
- **Constructs**: Evaluation of each CRITICAL construct is a unit of work. Each
  PARALLEL construct contained within a WORKSHARE construct is a single
  unit of work. New teams of threads execute the statements contained within the
  lexical extent of the enclosed PARALLEL constructs. In FORALL constructs or
  statements, the evaluation of the mask expression, expressions occurring in the
  specification of the iteration space, and the masked assignments are units of work.
  In WHERE constructs or statements, the evaluation of the mask expression and
  the masked assignments are units of work.
- **Directives**: The update of each scalar variable for an ATOMIC directive and its
  assignments is a unit of work.
- **ELEMENTAL functions**: If the argument to an ELEMENTAL function is an
  array, then the application of the function to each element of an array is a unit of
  work.

If none of the above definitions apply to a statement within the block, then that
statement is a unit of work.

**Rules**

In order to ensure that the statements within a WORKSHARE construct execute in
parallel, the construct must be enclosed within the dynamic extent of a parallel
region. Threads encountering a WORKSHARE construct outside the dynamic
extent of a parallel region will evaluate the statements within the construct serially.

A WORKSHARE directive binds to the closest dynamically enclosing PARALLEL
directive if one exists.

You must not nest DO, SECTIONS, SINGLE and WORKSHARE directives that
bind to the same PARALLEL directive

You must not specify a WORKSHARE directive within the dynamic extent of
CRITICAL, MASTER, or ORDERED directives.

You must not specify BARRIER, MASTER, or ORDERED directives within the
dynamic extent of a WORKSHARE construct.

If an array assignment, scalar assignment, a masked array assignment or a
FORALL assignment assigns to a private variable in the block, the result is
undefined.
If an array expression in the block references the value, association status or allocation status of private variables, the value of the expression is undefined unless each thread computes the same value.

If you do not specify a NO WAIT clause at the end of a WORKSHARE construct, a BARRIER directive is implied.

A WORKSHARE construct must be encountered by all threads in the team or by none at all.

Examples

Example 1: In the following example, the WORKSHARE directive evaluates the masked expressions in parallel.

```omply
$OMP WORKSHARE
   FORALL (I = 1 : N, AA(1, I) == 0) AA(1, I) = I
   BB = TRANSPOSE(AA)
   CC = MATMUL(AA, BB)
$OMP ATOMIC
   S = S + SUM(CC)
$OMP END WORKSHARE
```

Example 2: The following example includes a user defined ELEMENTAL as part of a WORKSHARE construct.

```omply
$OMP WORKSHARE
   WHERE (AA(1, :) /= 0.0) AA(1, :) = 1 / AA(1, :)
   DD = TRANS(AA(1, :))
$OMP END WORKSHARE

   ELEMENTAL REAL FUNCTION TRANS(ELM) RESULT(RES)
   REAL, INTENT(IN) :: ELM
   RES = ELM * ELM + 4
   END FUNCTION
```

Related reference

"ATOMIC" on page 67
"BARRIER" on page 70
"CRITICAL / END CRITICAL" on page 71
"PARALLEL WORKSHARE / END PARALLEL WORKSHARE" on page 93

See -qsmp in the Compiler Reference

OpenMP directive clauses

The OpenMP directive clauses allow you to specify additional information to directives.

The COLLAPSE, IF, NUM_THREADS, ORDERED, and SCHEDULE clauses allow you to control the parallel environment of a parallel region. See the detailed directive descriptions for more information.
Global rules for directive clauses

You must not specify a variable or common block name more than once in a clause.

A variable, common block name, or variable name that is a member of a common block must not appear in more than one clause on the same directive, with the following exceptions:

- You can define a named common block or named variable as `FIRSTPRIVATE` and `LASTPRIVATE` for the same directive.
- A variable appearing in a `NUM_THREADS` clause can appear in another clause for the same directive.
- A variable appearing in a `IF` clause can appear in another clause for the same directive.

If you do not specify a clause that changes the scope of a variable, the default scope for variables affected by a directive is `SHARED`.

A local variable with the `SAVE` or `STATIC` attribute declared in a procedure referenced within the dynamic extent of a parallel region has an implicit `SHARED` attribute. A local variable without the `SAVE` or `STATIC` attribute declared in a procedure referenced within the dynamic extent of a parallel region has an implicit `PRIVATE` attribute.

Members of common blocks and variables of modules declared in a procedure referenced within the dynamic extent of a parallel region have an implicit `SHARED` attribute, unless they are `THREADLOCAL` or `THREADPRIVATE`.

While a parallel or work-sharing construct is running, a variable or variable subobject used in a `PRIVATE`, `FIRSTPRIVATE`, `LASTPRIVATE` or `REDUCTION` clause of the directive must not be referenced, become defined, become undefined, have its association status or allocation status changed, or appear as an actual argument:

- In a scoping unit other than the one in which the directive construct appears
- In a variable format expression

You can declare a variable as `PRIVATE`, `FIRSTPRIVATE`, `LASTPRIVATE`, or `REDUCTION`, even if that variable is already storage associated with other variables. Storage association may exist for variables declared in `EQUIVALENCE` statements or in `COMMON` blocks. If a variable is storage associated with a `PRIVATE`, `FIRSTPRIVATE`, `LASTPRIVATE`, or `REDUCTION` variable, then:

- The contents, allocation status and association status of the variable that is storage associated with the `PRIVATE`, `FIRSTPRIVATE`, `LASTPRIVATE` or `REDUCTION` variable are undefined on entry to the parallel construct.
- The allocation status, association status and the contents of the associated variable become undefined if you define the `PRIVATE`, `FIRSTPRIVATE`, `LASTPRIVATE` or `REDUCTION` variable or if you define that variable’s allocation or association status.
- The allocation status, association status and the contents of the `PRIVATE`, `FIRSTPRIVATE`, `LASTPRIVATE` or `REDUCTION` variable become undefined if you define the associated variable or if you define the associated variable’s allocation or association status.
Pointers and OpenMP API Version 2.5

OpenMP API Version 2.5 allows a variable or variable subobject of a [PRIVATE] clause to have the POINTER or ALLOCATABLE attribute. The association status of the pointer is undefined at thread creation and when the thread is destroyed. The allocatable array must have an allocation status of “not currently allocated” on entry to and exit from OpenMP constructs.

XL Fortran provides an extension which allows a variable or variable subobject of a FIRSTPRIVATE or LASTPRIVATE clause to have the POINTER attribute. For FIRSTPRIVATE pointers at thread creation, each copy of the pointer receives the same association status as the original. If the pointer is used in a LASTPRIVATE clause, the pointer retains its association status at the end of the last iteration or SECTION.

To maintain full compliance with the OpenMP API Version 2.5 standard, ensure that a POINTER variable applies only to a PRIVATE clause.

COLLAPSE

Purpose

Specifying the COLLAPSE clause allows you to parallelize multiple loops in a nest without introducing nested parallelism.

Syntax

```
COLLAPSE (n)
```

$n$ is a positive constant integer expression

Rules

- Only one collapse clause is allowed on a worksharing DO or PARALLEL DO directive.
- The specified number of loops must be present lexically. That is, none of the loops can be in a called subroutine.
- The loops must form a rectangular iteration space and the bounds and stride of each loop must be invariant over all the loops.
- If the loop indices are of different size, the index with the largest size will be used for the collapsed loop.
- The loops must be perfectly nested; that is, there is no intervening code nor any OpenMP directive between the loops which are collapsed.
- The associated do-loops must be structured blocks. Their execution must not be terminated by an EXIT statement.
- If multiple loops are associated to the loop construct, only an iteration of the innermost associated loop may be curtailed by a CYCLE statement. If multiple loops are associated to the loop construct, there must be no branches to any of the loop termination statements except for the innermost associated loop.

Ordered construct

During execution of an iteration of a loop or a loop nested within a loop region, the executing thread must not execute more than one ordered region which binds to the same loop region. As a consequence, if multiple
loops are associated to the loop construct by a collapse clause, the ordered construct has to be located inside all associated loops.

LASTPRIVATE clause
When a LASTPRIVATE clause appears on the directive that identifies a work-sharing construct, the value of each new list item from the sequentially last iteration of the associated loops is assigned to the original list item even if a collapse clause is associated with the loop.

Other SMP and performance directives
The STREAM_UNROLL, UNROLL, UNROLL_AND_FUSE, and NOUNROLL_AND_FUSE directives cannot be used for any of the loops associated with the COLLAPSE clause loop nest. The INDEPENDENT directive can be used for any of the loops associated with the COLLAPSE clause.

Examples
In Example 1 and Example 2 the loops over k and j are collapsed and their iteration space is executed by all threads of the current team.

Example 1
```fortran
!$omp do collapse(2) private(i,j,k)
    do k = kl, ku, ks
        do j = jl, ju, js
            do i = il, iu, is
                call bar(a,i,j,k)
            enddo
        enddo
    enddo
end do
```

Example 2
```fortran
program test
    !$omp parallel
    !$omp do private(j,k) collapse(2) lastprivate(jlast, klast)
        do k = 1, 2
            do j = 1, 3
                jlast=j
                klast=k
            enddo
        enddo
    !$omp end do
    !$omp single
    print *, klast, jlast
    !$omp end single
    !$omp end parallel
end program test
```

Output:
2 3

Example 3
As both loops are collapsed into one, the ordered construct has to be inside all loops associated to the for construct. As an iteration may not execute more than one ordered region, this program would be incorrect without the collapse(2) clause.
program test
!$omp parallel num_threads(2)
!$omp do collapse(2) ordered private(j,k) schedule(static,3)
  do k = 1,3
    do j = 1,2
      !$omp ordered
      print *, k, j
      !$omp end ordered
      enddo
    enddo
  !$omp end do
!$omp end parallel
end program test

Output:

1 1
1 2
2 1
2 2
3 1
3 2

Related reference
DO
PARALLEL DO

COPYIN

Purpose

If you specify the COPYIN clause, the master thread’s copy of each variable, or common block declared in the copyin_entity_list is duplicated at the beginning of a parallel region. Each thread in the team that will execute within that parallel region receives a private copy of all entities in the copyin_entity_list. All variables declared in the copyin_entity_list must be THREADPRIVATE or members of a common block that appears in a THREADPRIVATE directive.

Syntax

COPYIN(copyin_entity_list)

copyin_entity

variable_name

/ common_block_name/

variable is a THREADPRIVATE variable, or THREADPRIVATE variable in a common block

common_block_name is a THREADPRIVATE common block name
Rules

If you specify a COPYIN clause, you cannot:
- specify the same entity name more than once in a copyin_entity_list.
- specify the same entity name in separate COPYIN clauses on the same directive.
- specify both a common block name and any variable within that same named common block in a copyin_entity_list.
- specify both a common block name and any variable within that same named common block in different COPYIN clauses on the same directive.
- specify a variable with the ALLOCATABLE attribute.

When the master thread of a team of threads reaches a directive containing the COPYIN clause, thread’s private copy of a variable or common block specified in the COPYIN clause will have the same value as the master thread’s copy.

On entry into any parallel region, a THREADPRIVATE variable, or a variable in a THREADPRIVATE common block is subject to the following criteria when declared in a COPYIN clause:
- If the variable has the POINTER attribute and the master thread’s copy of the variable is associated with a target, then each copy of that variable is associated with the same target. If the master thread’s pointer is disassociated, then each copy of that variable is disassociated. If the master thread’s copy of the variable has an undefined association status, then each copy of that variable has an undefined association status.
- Each copy of a variable without the POINTER attribute becomes defined with the value of the master thread’s copy as if by intrinsic assignment.

Related reference
PARALLEL
PARALLEL DO
PARALLEL SECTIONS
PARALLEL WORKSHARE

COPYPRIVATE

Purpose

If you specify the COPYPRIVATE clause, the value of a private variable or pointer to a shared object from one thread in a team is copied into the corresponding variables of all other threads in that team. If the variable in copyprivate_entity_list is not a pointer, then the corresponding variables of all threads within that team are defined with the value of that variable. If the variable is a pointer, then the corresponding variables of all threads within that team are defined with the association status of the pointer. Integer pointers and assumed-size arrays must not appear in copyprivate_entity_list.

Syntax

COPYPRIVATE—(copyprivate_entity_list—)
variable is a private variable within the enclosing parallel region

common_block_name is a THREADPRIVATE common block name

Rules

If a common block is part of the copyprivate_entity_list, then it must appear in a THREADPRIVATE directive. Furthermore, the COPYPRIVATE clause treats a common block as if all variables within its object_list were specified in the copyprivate_entity_list.

A COPYPRIVATE clause must occur on an END SINGLE directive at the end of a SINGLE construct. The compiler evaluates a COPYPRIVATE clause before any threads have passed the implied BARRIER directive at the end of that construct. The variables you specify in copyprivate_entity_list must not appear in a PRIVATE or FIRSTPRIVATE clause for the SINGLE construct. If the END SINGLE directive occurs within the dynamic extent of a parallel region, the variables you specify in copyprivate_entity_list must be private within that parallel region.

A COPYPRIVATE clause must not appear on the same END SINGLE directive as a NOWAIT clause.

A THREADLOCAL common block, or members of that common block, are not permitted as part of a COPYPRIVATE clause.

Related reference

END SINGLE

DEFAULT

Purpose

If you specify the DEFAULT clause, all variables in the lexical extent of the parallel construct will have a scope attribute of default_scope_attr.

If you specify DEFAULT(NONE), there is no default scope attribute. Therefore, you must explicitly list each variable you use in the lexical extent of the parallel construct in a data scope attribute clause on the parallel construct, unless the variable is:

• THREADPRIVATE
• A member of a THREADPRIVATE common block.
• A pointee
• A loop iteration variable used only as a loop iteration variable for:
  – Sequential loops in the lexical extent of the parallel region, or,
  – Parallel do loops that bind to the parallel region
A variable that is only used in work-sharing constructs that bind to the parallel region, and is specified in a data scope attribute clause for each of the work-sharing constructs.

The **DEFAULT** clause specifies that all variables in the parallel construct share the same default scope attribute of either **FIRSTPRIVATE**, **PRIVATE**, **SHARED**, or no default scope attribute.

**Syntax**

```
  DEFAULT--(default_scope_attr)--
```

*default_scope_attr* is one of **FIRSTPRIVATE**, **PRIVATE**, **SHARED**, or **NONE**

**Rules**

If you specify **DEFAULT(NONE)** on a directive you must specify all named variables and all the leftmost names of referenced array sections, array elements, structure components, or substrings in the lexical extent of the directive construct in a **FIRSTPRIVATE**, **LASTPRIVATE**, **PRIVATE**, **REDUCTION**, or **SHARED** clause.

If you specify **DEFAULT(FIRSTPRIVATE)** on a directive, all named variables and all leftmost names of referenced array sections, array elements, structure components, or substrings in the lexical extent of the directive construct, including common block and use associated variables, but excluding **POINTEE**s and **THREADLOCAL** common blocks, have a **FIRSTPRIVATE** attribute to a thread as if they were listed explicitly in a **FIRSTPRIVATE** clause.

If you specify **DEFAULT(PRIVATE)** on a directive, all named variables and all leftmost names of referenced array sections, array elements, structure components, or substrings in the lexical extent of the directive construct, including common block and use associated variables, but excluding **POINTEE**s and **THREADLOCAL** common blocks, have a **PRIVATE** attribute to a thread as if they were listed explicitly in a **PRIVATE** clause.

If you specify **DEFAULT(SHARED)** on a directive, all named variables and all leftmost names of referenced array sections, array elements, structure components, or substrings in the lexical extent of the directive construct, excluding **POINTEE**s have a **SHARED** attribute to a thread as if they were listed explicitly in a **SHARED** clause.

The default behavior will be **DEFAULT(SHARED)** if you do not explicitly indicate a **DEFAULT** clause on a directive.

**Examples**

The following example demonstrates the use of **DEFAULT(NONE)**, and some of the rules for specifying the data scope attributes of variables in the parallel region.
PROGRAM MAIN
COMMON /COMBLK/ ABC(10), DEF

! THE LOOP ITERATION VARIABLE, I, IS NOT
! REQUIRED TO BE IN DATA SCOPE ATTRIBUTE CLAUSE

!$OMP PARALLEL DEFAULT(NONE) SHARED(ABC)

! DEF IS SPECIFIED ON THE WORK-SHARING DO AND IS NOT
! REQUIRED TO BE SPECIFIED IN A DATA SCOPE ATTRIBUTE
! CLAUSE ON THE PARALLEL REGION.

!$OMP DO FIRSTPRIVATE(DEF)
DO I=1,10
   ABC(I) = DEF
END DO
!$OMP END PARALLEL
END

Related reference
PARALLEL
PARALLEL DO
PARALLEL SECTIONS
PARALLEL WORKSHARE
"TASK" on page 104

IF

Purpose

If you specify the IF clause, the runtime environment evaluates whether the scalar_logical_expression is true or false. If scalar_logical_expression is:

• true, the block is run in parallel.
• false, the containing region is suspended and the generated task is immediately run as though it is in a distinct task region.

Note that for the TASK directive, if the IF clause is evaluated to true, the block is not required to run in parallel.

Syntax

```
IF(—scalar_logical_expression—)  
```

Rules

The IF clause can be used in the PARALLEL, PARALLEL DO, PARALLEL SECTIONS, PARALLEL WORKSHARE, and TASK directives

The IF clause may appear at most once in any directive.

By default, a nested parallel loop is serialized, regardless of the setting of the IF clause. You can change this default by using the -qsmp=nested_par compiler option.

An IF expression is evaluated outside of the context of the parallel construct. Any function reference in the IF expression must not have side effects.
**FIRSTPRIVATE**

**Purpose**

If you use the `FIRSTPRIVATE` clause, each thread has its own initialized local copy of the variables and common blocks in `data_scope_entity_list`.

The `FIRSTPRIVATE` clause can be specified for the same variables as the `PRIVATE` clause, and functions in a manner similar to the `PRIVATE` clause. The exception is the status of the variable upon entry into the directive construct; the `FIRSTPRIVATE` variable exists and is initialized for each thread entering the directive construct.

**Syntax**

```
FIRSTPRIVATE—(—data_scope_entity_list—)
```

**Rules**

A variable in a `FIRSTPRIVATE` clause must not be any of the following elements:

- A pointee
- An assumed-size array
- A `THREADLOCAL` common block
- A `THREADPRIVATE` common block or its members
- A `THREADPRIVATE` variable
- An allocatable object

You cannot specify a variable in a `FIRSTPRIVATE` clause of a parallel construct if:

- the variable appears in a namelist statement, variable format expression or in an expression for a statement function definition, and,
- you reference the statement function, the variable format expression through formatted I/O, or the namelist through namelist I/O, within the parallel construct.

If one of the entities involved in an asynchronous I/O operation is a `FIRSTPRIVATE` variable, a subobject of a `FIRSTPRIVATE` variable, or a pointer that is associated with a `FIRSTPRIVATE` variable, the matching implied wait or `WAIT` statement must be executed before the end of the thread.

When individual members of a common block are privatized, the storage of the specified variable is no longer associated with the storage of the common block.
Any variable that is storage associated with a FIRSTPRIVATE variable is undefined on entrance into the parallel construct.

If a directive construct contains a FIRSTPRIVATE argument to a Message Passing Interface (MPI) routine performing non-blocking communication, the MPI communication must complete before the end of the construct.

Related reference

DO
PARALLEL
PARALLEL DO
PARALLEL SECTIONS
PARALLEL WORKSHARE
SECTIONS
SINGLE

"TASK" on page 104

LASTPRIVATE

Purpose

If you use the LASTPRIVATE clause, each variable and common block in data_scope_entity_list is PRIVATE, and the last value of each variable in data_scope_entity_list can be referred to outside of the construct of the directive. If you use the LASTPRIVATE clause with DO or PARALLEL DO, the last value is the value of the variable after the last sequential iteration of the loop. If you use the LASTPRIVATE clause with SECTIONS or PARALLEL SECTIONS, the last value is the value of the variable after the last SECTION of the construct. If the last iteration of the loop or last section of the construct does not define a LASTPRIVATE variable, the variable is undefined after the loop or construct.

The LASTPRIVATE clause functions in a manner similar to the PRIVATE clause and you should specify it for variables that match the same criteria. The exception is in the status of the variable on exit from the directive construct. The compiler determines the last value of the variable, and takes a copy of that value which it saves in the named variable for use after the construct. A LASTPRIVATE variable is undefined on entry to the construct if it is not a FIRSTPRIVATE variable.

Syntax

LASTPRIVATE—(—data_scope_entity_list—)

Rules

A variable in a LASTPRIVATE clause must not be any of the following elements:

- A pointee
- An allocatable object
- An assumed-size array
- A THREADLOCAL common block
- A THREADPRIVATE common block or its members
- A THREADPRIVATE variable
You cannot specify a variable in a `LASTPRIVATE` clause of a parallel construct if:
- the variable appears in a namelist statement, variable format expression or in an expression for a statement function definition, and,
- you reference the statement function, the variable format expression through formatted I/O, or the namelist through namelist I/O, within the parallel construct.

If one of the entities involved in an asynchronous I/O operation is a `LASTPRIVATE`, a subobject of a `LASTPRIVATE` variable, or a pointer that is associated with a `LASTPRIVATE` variable, the matching implied wait or `WAIT` statement must be executed before the end of the thread.

When individual members of a common block are privatized, the storage of the specified variable is no longer associated with the storage of the common block.

Any variable that is storage associated with a `LASTPRIVATE` variable is undefined on entrance into the parallel construct.

If a directive construct contains a `LASTPRIVATE` argument to a Message Passing Interface (MPI) routine performing non-blocking communication, the MPI communication must complete before the end of that construct.

If you specify a variable as `LASTPRIVATE` on a work-sharing directive, and you have specified a `NOWAIT` clause on that directive, you cannot use that variable between the end of the work-sharing construct and a `BARRIER`.

Variables that you specify as `LASTPRIVATE` to a parallel construct become defined at the end of the construct. If you have concurrent definitions or uses of `LASTPRIVATE` variables on multiple threads, you must ensure that the threads are synchronized at the end of the construct when the variables become defined. For example, if multiple threads encounter a `PARALLEL` construct with a `LASTPRIVATE` variable, you must synchronize the threads when they reach the `END PARALLEL` directive, because the `LASTPRIVATE` variable becomes defined at `END PARALLEL`. Therefore the whole `PARALLEL` construct must be enclosed within a synchronization construct.

**Examples**

The following example shows the proper use of a `LASTPRIVATE` variable after a `NOWAIT` clause.
```
!$OMP PARALLEL
!$OMP DO LASTPRIVATE(K)
    DO I=1,10
       K=I+1
    END DO
!$OMP END DO NOWAIT
! PRINT *, K **ERROR** ! The reference to K must occur after a ! barrier.
!$OMP BARRIER
! PRINT *, K ! This reference to K is legal.
!$OMP END PARALLEL
END
```

**Related reference**
NUM_THREADS

Purpose
The NUM_THREADS clause allows you to specify the number of threads used in a parallel region. Subsequent parallel regions are not affected. The NUM_THREADS clause takes precedence over the number of threads specified using the omp_set_num_threads library routine or the environment variable OMP_NUM_THREADS.

Syntax

```fortran
NUM_THREADS( scalar_integer_expression )
```

Rules
The value of scalar_integer_expression must be a positive. Evaluation of the expression occurs outside the context of the parallel region. Any function calls that appear in the expression and change the value of a variable referenced in the expression will have unspecified results.

If you are using the environment variable OMP_DYNAMIC to enable dynamic threads, scalar_integer_expression defines the maximum number of threads available in the parallel region.

You must specify the omp_set_nested library routine or set the OMP_NESTED environment variable when including the NUM_THREADS clause as part of a nested parallel regions otherwise, the execution of that parallel region is serialized.

Related reference

ORDERED

Purpose
Specifying the ORDERED clause on a work-sharing construct allows you to specify the ORDERED directive within the dynamic extent of a parallel loop.

Syntax

```fortran
ORDERED
```
Rules

The ORDERED clause applies to the following directives:

Related reference

“DO / END DO” on page 73
“PARALLEL DO / END PARALLEL DO” on page 86

PRIVATE

Purpose

If you specify the PRIVATE clause on one of the directives listed below, each thread in a team has its own uninitialized local copy of the variables and common blocks in data_scope_entity_list.

You should specify a variable with the PRIVATE attribute if its value is calculated by a single thread and that value is not dependent on any other thread, if it is defined before it is used in the construct, and if its value is not used after the construct ends. Copies of the PRIVATE variable exist, locally, on each thread. Each thread receives its own uninitialized copy of the PRIVATE variable. A PRIVATE variable has an undefined value or association status on entry to, and exit from, the directive construct. All thread variables within the lexical extent of the directive construct have the PRIVATE attribute by default.

Syntax

```
PRIVATE—(data_scope_entity_list)
```

Rules

A variable in the PRIVATE clause must not be any of the following elements:

- A pointee
- An assumed-size array
- A THREADLOCAL common block
- A THREADPRIVATE common block or its members
- A THREADPRIVATE variable or the variable equivalenced with a THREADPRIVATE variable

You cannot specify a variable in a PRIVATE clause of a parallel construct if:

- the variable appears in a namelist statement, variable format expression or in an expression for a statement function definition, and,
- you reference the statement function, the variable format expression through formatted I/O, or the namelist through namelist I/O, within the parallel construct.

If one of the entities involved in an asynchronous I/O operation is a PRIVATE variable, a subobject of a PRIVATE variable, or a pointer that is associated with a PRIVATE variable, the matching implied wait or WAIT statement must be executed before the end of the thread.
When individual members of a common block are privatized, the storage of the specified variable is no longer associated with the storage of the common block.

Any variable that is storage associated with a PRIVATE variable is undefined on entrance into the parallel construct.

If a directive construct contains a PRIVATE argument to a Message Passing Interface (MPI) routine performing non-blocking communication, the MPI communication must complete before the end of that construct.

A variable name in the data_scope_entity_list of the PRIVATE clause can be an allocatable object. It must not be allocated on initial entry to the directive construct, and you must allocate and deallocate the object for every thread that executes the construct.

Local variables without the SAVE or STATIC attributes in referenced subprograms in the dynamic extent of a directive construct have an implicit PRIVATE attribute.

Examples

The following example demonstrates the proper use of a PRIVATE variable that is used to define a statement function. A commented line shows the invalid use.

Since J appears in a statement function, the statement function cannot be referenced within the parallel construct for which J is PRIVATE.

```fortran
INTEGER :: ARR(10), J = 17
ISTFNC() = J

!$OMP PARALLEL PRIVATE(J)
DO I = 1, 10
   J=I
   ARR(I) = J
   ! ARR(I) = ISTFNC() **ERROR**  A reference to ISTFNC would
   ! make the PRIVATE(J) clause
   ! invalid.
END DO
PRINT *, ARR
END
```

Related reference

DO
PARALLEL
PARALLEL DO
PARALLEL SECTIONS
PARALLEL WORKSHARE
SECTIONS
SINGLE
"TASK" on page 104

REDUCTION

Purpose

The REDUCTION clause updates named variables declared on the clause within the directive construct. Intermediate values of REDUCTION variables are not used within the parallel construct other than in the updates themselves.
op_fnc is a reduction_op or a reduction_function that appears in all REDUCTION statements involving this variable. You must not specify more than one REDUCTION operator or function for a variable in the directive construct. To maintain OpenMP API Version 2.5 compliance, you must specify op_fnc for the REDUCTION clause.

A REDUCTION statement can have one of the following forms:

\[
\begin{align*}
\text{REDUCTION} & \left( \text{variable_name_list} \right) \text{op_fnc} : \end{align*}
\]

where:

- \text{reduction_var_ref} is a variable or subobject of a variable that appears in a REDUCTION clause.
- \text{reduction_op} is one of the intrinsic operators: +, -, *, .AND., .OR., .EQV., .NEQV., or .XOR.
  - when \text{reduction_op} is an intrinsic operator, it should be the last operation performed on the right side.
- \text{reduction_function} is one of the intrinsic procedures: MAX, MIN, IAND, IOR, or IEOR.
- \text{expr} should not contain references to \text{reduction_var_ref}.

The canonical initialization value of each of the operators and intrinsics are shown in the following table. The actual initialization value will be consistent with the data type of your corresponding REDUCTION variable.

<table>
<thead>
<tr>
<th>Intrinsic Operator</th>
<th>Initialization</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td>*</td>
<td>1</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>.AND.</td>
<td>.TRUE.</td>
</tr>
<tr>
<td>.OR.</td>
<td>.FALSE.</td>
</tr>
</tbody>
</table>

---

130  XL Fortran: Optimization and Programming Guide
### Rules

The following rules apply to **REDUCTION** statements:

- A variable in the **REDUCTION** clause must only occur in a **REDUCTION** statement within the directive construct on which the **REDUCTION** clause appears.
- The two `reduction_var_ref` that appear in a **REDUCTION** statement must be lexically identical.
- You cannot use the following form of the **REDUCTION** statement:
  ```fortran
  reduction_var_ref = expr operator reduction_var_ref, where operator is any operator other than reduction_op.
  ```

When you specify individual members of a common block in a **REDUCTION** clause, the storage of the specified variable is no longer associated with the storage of the common block.

Any variable you specify in a **REDUCTION** clause of a work-sharing construct must be shared in the enclosing **PARALLEL** construct.

If you use a **REDUCTION** clause on a construct that has a **NOWAIT** clause, the **REDUCTION** variable remains undefined until a barrier synchronization has been performed to ensure that all threads have completed the **REDUCTION** clause.

A **REDUCTION** variable must not appear in a **FIRSTPRIVATE**, **PRIVATE** or **LASTPRIVATE** clause of another construct within the dynamic extent of the construct in which it appeared as a **REDUCTION** variable.

If you specify `op_fnc` for the **REDUCTION** clause, each variable in the `variable_name_list` must be of intrinsic type. The variable can only appear in a **REDUCTION** statement within the lexical extent of the directive construct. You must specify `op_fnc` if the directive uses the `trigger_constant` **SOMP**.

The **REDUCTION** clause specifies named variables that appear in reduction operations. The compiler will maintain local copies of such variables, but will combine them upon exit from the construct. The intermediate values of the **REDUCTION** variables are combined in random order, dependent on which threads finish their calculations first. Therefore, there is no guarantee that bit-identical results will be obtained from one parallel run to another. This is true even if the parallel runs use the same number of threads, scheduling type, and chunk size.
Variables that you specify as REDUCTION or LASTPRIVATE to a parallel construct become defined at the end of the construct. If you have concurrent definitions or uses of REDUCTION or LASTPRIVATE variables on multiple threads, you must ensure that the threads are synchronized at the end of the construct when the variables become defined. For example, if multiple threads encounter a PARALLEL construct with a REDUCTION variable, you must synchronize the threads when they reach the END PARALLEL directive, because the REDUCTION variable becomes defined at END PARALLEL. Therefore the whole PARALLEL construct must be enclosed within a synchronization construct.

A variable in the REDUCTION clause must be of intrinsic type. A variable in the REDUCTION clause, or any element thereof, must not be any of the following:

- A pointee
- An assumed-size array
- A THREADLOCAL common block
- A THREADPRIVATE common block or its members
- A THREADPRIVATE variable
- An Allocatable object
- A Fortran 90 pointer

These rules describe the use of REDUCTION on OpenMP directives. If you are using the REDUCTION clause on the INDEPENDENT directive, see the INDEPENDENT directive in the XL Fortran Language Reference directive.

Related reference

- DO
- PARALLEL
- PARALLEL DO
- PARALLEL SECTIONS
- PARALLEL WORKSHARE
- SECTIONS

SCHEDULE

Purpose

The SCHEDULE clause allows you to specify the chunking method for parallelization. Work is assigned to threads in different manners depending on the scheduling type or chunk size used.

Syntax

```
SCHEDULE(-(sched_type){n})
```

sched_type

is one of AFFINITY, DYNAMIC, GUIDED, RUNTIME, or STATIC

n

must be a positive scalar integer expression; it must not be specified for the RUNTIME sched_type. If you are using the trigger_constant SOMP, do not specify the scheduling type AFFINITY.
AFFINITY
The iterations of a loop are initially divided into $number_of_threads$
partitions, containing
$\text{CEILING}(number_of_iterations / number_of_threads)$

iterations. Each partition is initially assigned to a thread, and is then
further subdivided into chunks containing $n$ iterations, if $n$ has been
specified. If $n$ has not been specified, then the chunks consist of
$\text{CEILING}(number_of_iterations\_remaining\_in\_partition / 2)$

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 that is initially
assigned to another thread.

Threads that are active will complete the work in a partition that is initially
assigned to a sleeping thread.

DYNAMIC
If $n$ has been specified, the iterations of a loop are divided into chunks
containing $n$ iterations each. If $n$ has not been specified, then the default
chunk size is 1 iteration.

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 other thread becomes available.

GUIDED
If you specify a value for $n$, the iterations of a loop are divided into chunks
such that the size of each successive chunk is exponentially decreasing. $n$
specifies the size of the smallest chunk, except possibly the last. If you do
not specify a value for $n$, the default value is 1.

The size of the initial chunk is proportional to:
$\text{CEILING}(number_of_iterations / number_of_threads)$

iterations. Subsequent chunks are proportional to:
$\text{CEILING}(number_of_iterations\_remaining / number_of_threads)$

iterations. If $n$ is greater than 1, each chunk should contain fewer than $n$
iterations (except for the last chunk to be assigned, which can have fewer
than $n$ iterations. As each thread finishes a chunk, it dynamically obtains
the next available chunk.

You can use guided scheduling in a situation in which multiple threads in
a team might arrive at a DO work-sharing construct at varying times, and
each iteration requires roughly the same amount of work. For example, if
you have a DO loop preceded by one or more work-sharing SECTIONS
or DO constructs with NOWAIT clauses, you can guarantee that no thread
waits at the barrier longer than it takes another thread to execute its final
iteration, or final $k$ iterations if a chunk size of $k$ is specified. The GUIDED
schedule requires the fewest synchronizations of all the scheduling
methods.
An \( n \) expression is evaluated outside of the context of the \texttt{DO} construct. Any function reference in the \( n \) expression must not have side effects.

The value of the \( n \) parameter on the \texttt{SCHEDULE} clause must be the same for all of the threads in the team.

**RUNTIME**

Determine the scheduling type at run time.

At run time, the scheduling type can be specified using the environment variable \texttt{XLSMPOPTS}. If no scheduling type is specified using that variable, then the default scheduling type used is \texttt{STATIC}.

**STATIC**

If \( n \) has been specified, the iterations of a loop are divided into chunks that contain \( n \) iterations. Each thread is assigned chunks in a “round robin” fashion. This is known as block cyclic scheduling. If the value of \( n \) is 1, then the scheduling type is specifically referred to as cyclic scheduling.

If \( n \) has not been specified, the chunks will contain
\[
\text{CEILING}(\text{number_of_iterations} / \text{number_of_threads})
\]
iterations. Each thread is assigned one of these chunks. This is known as block cyclic scheduling.

If a thread is asleep and it has been assigned work, it will be awakened so that it may complete its work.

\texttt{STATIC} is the default scheduling type if the user has not specified any scheduling type at compile-time or run time.

**Rules**

You must not specify the \texttt{SCHEDULE} clause more than once for a particular \texttt{DO} directive.

**Related reference**

“\texttt{DO / END DO}” on page 73

“\texttt{PARALLEL DO / END PARALLEL DO}” on page 86

**SHARED**

**Purpose**

All sections use the same copy of the variables and common blocks you specify in \texttt{data_scope_entity_list}.

The \texttt{SHARED} clause specifies variables that must be available to all threads. If you specify a variable as \texttt{SHARED}, you are stating that all threads can safely share a single copy of the variable.

**Syntax**

```
$\texttt{SHARED}(\text{-data_scope_entity_list-})$
```

\texttt{data_scope_entity}
named_variable

is a named variable that is accessible in the directive construct

common_block_name

is a common block name that is accessible in the directive construct

Rules

A variable in the SHARED clause must not be either:

- A pointee
- A THREADLOCAL common block.
- A THREADPRIVATE common block or its members.
- A THREADPRIVATE variable.

If a SHARED variable, a subobject of a SHARED variable, or an object associated with a SHARED variable or subobject of a SHARED variable appears as an actual argument in a reference to a non-intrinsic procedure and:

- The actual argument is an array section with a vector subscript; or
- The actual argument is
  - An array section,
  - An assumed-shape array, or,
  - A pointer array

and the associated dummy argument is an explicit-shape or assumed-size array;

then any references to or definitions of the shared storage that is associated with the dummy argument by any other thread must be synchronized with the procedure reference. In other words, you must structure your code in such a way that if a thread encounters a procedure reference, then the procedure call by that thread and any reference to or definition of the shared storage by any other thread will always occur in the same sequence. You can do this, for example, by placing the procedure reference after a BARRIER.

Examples

In the following example, the procedure reference with an array section actual argument is required to be synchronized with references to the dummy argument by placing the procedure reference in a critical section, because the associated dummy argument is an explicit-shape array.

```fortran
INTEGER :: ABC(10)
I=2; J=5
!$OMP PARALLEL DEFAULT(NONE), SHARED(ABC,I,J)
!$OMP CRITICAL
    CALL SUB1(ABC(I:J)) ! ACTUAL ARGUMENT IS AN ARRAY
    ! SECTION; THE PROCEDURE
    ! REFERENCE MUST BE IN A CRITICAL SECTION.
!$OMP END CRITICAL
!$OMP END PARALLEL
```

Chapter 7. Parallel programming with XL Fortran 135
CONTAINS
SUBROUTINE SUB1(ARR)
INTEGER:: ARR(1:4)
DO I=1, 4
ARR(I) = I
END DO
END SUBROUTINE
END

Related reference
PARALLEL
PARALLEL DO
PARALLEL SECTIONS
PARALLEL WORKSHARE
"TASK" on page 104

UNTIED

Purpose

The UNTIED clause is used with the TASK directive. When a task region is suspended, untied tasks can be resumed by any thread in a team.

Syntax

UNTIED

Related reference
"TASK" on page 104

OpenMP execution environment, lock and timing routines (IBM extension)

The OpenMP specification provides a number of routines which allow you to control and query the parallel execution environment.

Parallel threads created by the runtime environment through the OpenMP interface are considered independent of the threads you create and control using calls to the Fortran Pthreads library module. References within the following descriptions to "serial portions of the program" refer to portions of the program that are executed by only one of the threads that have been created by the runtime environment. For example, you can create multiple threads by using f_pthread_create. However, if you then call omp_get_num_threads from outside of an OpenMP parallel block, or from within a serialized nested parallel region, the function will return 1, regardless of the number of threads that are currently executing.

OpenMP runtime library calls must not appear in PURE and ELEMENTAL procedures.
Table 15. OpenMP execution environment routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>omp_get_dynamic</td>
<td>omp_get_thread_num</td>
</tr>
<tr>
<td>omp_get_max_threads</td>
<td>omp_in_parallel</td>
</tr>
<tr>
<td>omp_get_nested</td>
<td>omp_set_dynamic</td>
</tr>
<tr>
<td>omp_get_num_procs</td>
<td>omp_set_nested</td>
</tr>
<tr>
<td>omp_get_num_threads</td>
<td>omp_set_num_threads</td>
</tr>
</tbody>
</table>

Included in the OpenMP runtime library are two routines that support a portable wall-clock timer.

Table 16. OpenMP timing routines

<table>
<thead>
<tr>
<th>Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>omp_get_wtick</td>
</tr>
<tr>
<td>omp_get_wtime</td>
</tr>
</tbody>
</table>

The OpenMP runtime library also supports a set of simple and nestable lock routines. You must only lock variables through these routines. Simple locks may not be locked if they are already in a locked state. Simple lock variables are associated with simple locks and may only be passed to simple lock routines. Nestable locks may be locked multiple times by the same thread. Nestable lock variables are associated with nestable locks and may only be passed to nestable lock routines.

For all the routines listed below, the lock variable is an integer whose KIND type parameter is denoted either by the symbolic constant omp_lock_kind, or by omp_nest_lock_kind.

This variable is sized according to the compilation mode. It is set either to ‘4’ for 32-bit applications or ‘8’ for 64-bit.

Table 17. OpenMP simple lock routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>omp_destroy_lock</td>
<td>omp_test_lock</td>
</tr>
<tr>
<td>omp_init_lock</td>
<td>omp_unset_lock</td>
</tr>
<tr>
<td>omp_set_lock</td>
<td></td>
</tr>
</tbody>
</table>

Table 18. OpenMP nestable lock routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>omp_destroy_nest_lock</td>
<td>omp_test_nest_lock</td>
</tr>
<tr>
<td>omp_init_nest_lock</td>
<td>omp_unset_nest_lock</td>
</tr>
<tr>
<td>omp_set_nest_lock</td>
<td></td>
</tr>
</tbody>
</table>

Note: You can define and implement your own versions of the OpenMP routines. However, by default, the compiler will substitute the XL Fortran versions of the OpenMP routines regardless of the existence of other implementations, unless you specify the -qnoswapomp compiler option. For more information, see XL Fortran Compiler Reference.
omp_destroy_lock(svar)

Purpose

This subroutine disassociates a given lock variable from all locks. You must use omp_init_lock to reinitialize a lock variable that was destroyed with a call to omp_destroy_lock before using it again as a lock variable.

If you call omp_destroy_lock with an uninitialized lock variable, the result of the call is undefined.

Class

Subroutine.

Argument Type and Attributes

svar  Type integer with kind omp_lock_kind.

Result Type and Attributes

None.

Result Value

None.

Examples

In the following example, one at a time, the threads gain ownership of the lock associated with the lock variable LCK, print the thread ID, and then release ownership of the lock.

```fortran
USE omp_lib
INTEGER(kind=omp_lock_kind) LCK
INTEGER ID
CALL omp_init_lock(LCK)
!$OMP PARALLEL SHARED(LCK), PRIVATE(ID)
   ID = omp_get_thread_num()
   CALL omp_set_lock(LCK)
   PRINT *,"MY THREAD ID IS", ID
   CALL omp_unset_lock(LCK)
!$OMP END PARALLEL
CALL omp_destroy_lock(LCK)
END
```

omp_destroy_nest_lock(nvar)

Purpose

This subroutine initializes a nestable lock variable, causing the lock variable to become undefined. The variable nvar must be an unlocked and initialized nestable lock variable.

If you call omp_destroy_nest_lock using an uninitialzied variable, the result is undefined.

Class

Subroutine.
Argument Type and Attributes
svar    Type integer with kind omp_nest_lock_kind.

Result Type and Attributes
None.

Result Value
None.

`omp_get_dynamic()`

Purpose
The `omp_get_dynamic` function returns .TRUE. if dynamic thread adjustment by the runtime environment is enabled. Otherwise, the `omp_get_dynamic` function returns .FALSE.

Class
Function.

Argument Type and Attributes
None.

Result Type and Attributes
Default logical.

Result Value
.

`omp_get_max_threads()`

Purpose
This function returns the maximum number of threads that can execute concurrently in a single parallel region. The return value is equal to the maximum value that can be returned by the `omp_get_num_threads` function. If you use `omp_set_num_threads` to change the number of threads, subsequent calls to `omp_get_max_threads` will return the new value.

The function has global scope, which means that the maximum value it returns applies to all functions, subroutines, and compilation units in the program. It returns the same value whether executing from a serial or parallel region.

You can use `omp_get_max_threads` to allocate maximum-sized data structures for each thread when you have enabled dynamic thread adjustment by passing `omp_set_dynamic` an argument which evaluates to .TRUE.
Class
Function.

Argument Type and Attributes
None.

Result Type and Attributes
Default integer.

Result Value
The maximum number of threads that can execute concurrently in a single parallel region.

**omp_get_nested()**

Purpose
The *omp_get_nested* function returns **.TRUE.** if nested parallelism is enabled and **.FALSE.** if nested parallelism is disabled.

Currently, XL Fortran does not support OpenMP nested parallelism.

Class
Function

Argument Type and Attributes
None.

Result Type and Attributes
Default logical.

Result Value
**.TRUE.** if nested parallelism is enabled. **.FALSE.** otherwise.

**omp_get_num_procs()**

Purpose
The *omp_get_num_procs* function returns the number of online processors on the machine.

Class
Function.

Argument Type and Attributes
None.
Result Type and Attributes

Default integer.

Result Value

The number of online processors on the machine.

**omp_get_num_threads()**

Purpose

The `omp_get_num_threads` function returns the number of threads in the team currently executing the parallel region from which it is called. The function binds to the closest enclosing PARALLEL directive.

The `omp_set_num_threads` subroutine and the OMP_NUM_THREADS environment variable control the number of threads in a team. If you do not explicitly set the number of threads, the runtime environment will use the number of online processors on the machine by default.

If you call `omp_get_num_threads` from a serial portion of your program or from a nested parallel region that is serialized, the function returns 1.

Class

Function.

Argument Type and Attributes

None.

Result Type and Attributes

Default integer.

Result Value

The number of threads in the team currently executing the parallel region from which the function is called.

Examples

```fortran
USE omp_lib
INTEGER N1, N2

N1 = omp_get_num_threads()
PRINT *, N1
!$OMP PARALLEL PRIVATE(N2)
N2 = omp_get_num_threads()
PRINT *, N2
!$OMP END PARALLEL
END
```

The `omp_get_num_threads` call returns 1 in the serial section of the code, so N1 is assigned the value 1. N2 is assigned the number of threads in the team executing the parallel region, so the output of the second print statement will be an arbitrary number less than or equal to the value returned by `omp_get_max_threads`. 
omp_get_thread_num()

Purpose

This function returns the number of the currently executing thread within the team. The number returned will always be between 0 and NUM_PARTHDS - 1. NUM_PARTHDS is the number of currently executing threads within the team. The master thread of the team returns a value of 0.

If you call omp_get_thread_num from within a serial region, from within a serialized nested parallel region, or from outside the dynamic extent of any parallel region, this function will return a value of 0.

This function binds to the closest parallel region.

Class

Function.

Argument Type and Attributes

None.

Result Type and Attributes

Default integer.

Result Value

The value of the currently executing thread within the team between 0 and NUM_PARTHDS - 1. NUM_PARTHDS is the number of currently executing threads within the team. A call to omp_get_thread_num from a serialized nested parallel region, or from outside the dynamic extent of any parallel region returns 0.

Examples

The following example illustrates the return value of the omp_get_thread_num routine in a PARALLEL region and a MASTER construct.

```fortran
USE omp_lib
INTEGER NP
CALL omp_set_num_threads(4)  ! 4 threads are used in the parallel region

 !$OMP PARALLEL PRIVATE(NP)
 NP = omp_get_thread_num()
 CALL WORK('in parallel', NP)
 !$OMP END PARALLEL

 !$OMP MASTER
 NP = omp_get_thread_num()
 CALL WORK('in master', NP)
 !$OMP END MASTER
 !$OMP END PARALLEL
END SUBROUTINE WORK(msg, THD_NUM)
INTEGER THD_NUM
character(*) msg
PRINT *, msg, THD_NUM
END
```
Output:
in parallel 1
in parallel 3
in parallel 2
in parallel 0
in master 0

(The order may be different.)

**omp_get_wtick()**

**Purpose**

The `omp_get_wtick` function returns a double precision value equal to the number of seconds between consecutive clock ticks.

**Class**

Function.

**Argument Type and Attributes**

None.

**Result Type and Attributes**

Double precision real.

**Result Value**

The number of seconds between consecutive ticks of the operating system real-time clock.

**Examples**

```fortran
USE omp_lib
DOUBLE PRECISION WTICKS
WTICKS = omp_get_wtick()
PRINT *, 'The clock ticks ', 10 / WTICKS, &
' times in 10 seconds.'
END
```

**omp_get_wtime()**

**Purpose**

The `omp_get_wtime` function returns a double precision value equal to the number of seconds since the initial value of the operating system real-time clock. The initial value is guaranteed not to change during execution of the program.

The value returned by the `omp_get_wtime` function is not consistent across all threads in the team.

**Class**

Function.
Argument Type and Attributes

None.

Result Type and Attributes

Double precision real.

Result Value

The number of seconds since the initial value of the operating system real-time clock.

Examples

USE omp_lib
DOUBLE PRECISION START, END
START = omp_get_wtime()
! Work to be timed
END = omp_get_wtime()
PRINT *, 'Stuff took ', END - START, ' seconds.'
END

omp_in_parallel()

Purpose

The omp_in_parallel function returns .TRUE. if you call it from the dynamic extent of a region executing in parallel and returns .FALSE. otherwise. If you call omp_in_parallel from a region that is serialized but nested within the dynamic extent of a region executing in parallel, the function will still return .TRUE.. (Nested parallel regions are serialized by default. See “omp_set_nested(enable_expr)” on page 148 and the OMP_NESTED environment variable for more information.)

Class

Function.

Argument Type and Attributes

None.

Result Type and Attributes

Default logical.

Result Value

.FALSE. if called from the dynamic extent of a region executing in parallel. .TRUE. otherwise.

Examples

In the following example, the first call to omp_in_parallel returns .FALSE. because the call is outside the dynamic extent of any parallel region. The second call returns .TRUE., even if the nested PARALLEL DO loop is serialized, because the call is still inside the dynamic extent of the outer PARALLEL DO loop.
USE omp_lib
INTEGER N, M
N = 4
M = 3
PRINT*, omp_in_parallel()
 !$OMP PARALLEL DO
    DO I = 1, N
      !$OMP PARALLEL DO
        DO J=1, M
          PRINT *, omp_in_parallel()
        END DO
      !$OMP END PARALLEL DO
    END DO
 !$OMP END PARALLEL DO
END

omp_init_lock(svar)

Purpose

The **omp_init_lock** subroutine initializes a lock and associates it with the lock variable passed in as a parameter. After the call to **omp_init_lock**, the initial state of the lock variable is unlocked.

If you call this routine with a lock variable that you have already initialized, the result of the call is undefined.

**Class**

Subroutine.

**Argument Type and Attributes**

*svar*  Integer of kind **omp_lock_kind**.

**Result Type and Attributes**

None.

**Result Value**

None.

**Examples**

In the following example, one at a time, the threads gain ownership of the lock associated with the lock variable LCK, print the thread ID, and release ownership of the lock.

USE omp_lib
 INTEGER(kind=omp_lock_kind) LCK
 INTEGER ID
 CALL omp_init_lock(LCK)
 !$OMP PARALLEL SHARED(LCK), PRIVATE(ID)
    ID =omp_get_thread_num()
    CALL omp_set_lock(LCK)
    PRINT *, 'MY THREAD ID IS', ID
    CALL omp_unset_lock(LCK)
 !$OMP END PARALLEL
 CALL omp_destroy_lock(LCK)
END
omp_init_nest_lock(nvar)

Purpose

The `omp_init_nest_lock` subroutine allows you to initialize a nestable lock and associate it with the lock variable you specify. The initial state of the lock variable is unlocked, and the initial nesting count is zero. The value of `nvar` must be an uninitialized nestable lock variable.

If you call `omp_init_nest_lock` using a variable that is already initialized, the result is undefined.

Class

Subroutine.

Argument Type and Attributes

nvar  Integer of kind `omp_nest_lock_kind`.

Result Type and Attributes

None.

Result Value

None.

Examples

The following example to illustrate the use of nestable lock for updating variable P in the PARALLEL SECTIONS construct.

```fortran
USE omp_lib
INTEGER P
INTEGER A
INTEGER B
INTEGER ( kind=omp_nest_lock_kind ) LCK
CALL omp_init_nest_lock ( LCK )  ! initialize the nestable lock
!
!$OMP PARALLEL SECTIONS
!$OMP SECTION
CALL omp_set_nest_lock ( LCK )
P = P + A
CALL omp_set_nest_lock ( LCK )
P = P + B
CALL omp_unset_nest_lock ( LCK )
CALL omp_unset_nest_lock ( LCK )
!$OMP SECTION
CALL omp_set_nest_lock ( LCK )
P = P + B
CALL omp_unset_nest_lock ( LCK )
!$OMP END PARALLEL SECTIONS

CALL omp_destroy_nest_lock ( LCK )
END
```
omp_set_dynamic(enable_expr)

Purpose

The `omp_set_dynamic` subroutine enables or disables dynamic adjustment, by the runtime environment, of the number of threads available to execute parallel regions.

If `enable_expr` is evaluated to `.TRUE.`, the runtime environment can automatically adjust the number of threads that are used to execute subsequent parallel regions to obtain the best use of system resources. The number of threads you specify using `omp_set_num_threads` becomes the maximum, not exact, thread count.

If `enable_expr` is evaluated to `.FALSE.`, dynamic adjustment of the number of threads is disabled. The runtime environment cannot automatically adjust the number of threads used to execute subsequent parallel regions. The value you pass to `omp_set_num_threads` becomes the exact thread count.

By default, dynamic thread adjustment is enabled. If your code depends on a specific number of threads for correct execution, you should explicitly disable dynamic threads.

If the routine is called from a portion of the program where the `omp_in_parallel` routine returns `.TRUE.`, the routine has no effect.

This subroutine has precedence over the `OMP_DYNAMIC` environment variable.

Class

Subroutine.

Argument Type and Attributes

`enable_expr`

Logical.

Result Type and Attributes

None.

Result Value

None.

omp_set_lock(svar)

Purpose

The `omp_set_lock` subroutine forces the calling thread to wait until the specified lock is available before executing subsequent instructions. The calling thread is given ownership of the lock when it becomes available.

If you call this routine with an uninitialized lock variable, the result of the call is undefined. If a thread that owns a lock tries to lock it again by issuing a call to `omp_set_lock`, the thread produces a deadlock.
Class
Subroutine.

Argument Type and Attributes
svar  Integer of kind omp_lock_kind.

Result Type and Attributes
None.

Result Value
None.

Examples
In the following example, the lock variable LCK_X is used to avoid race conditions when updating the shared variable X. By setting the lock before each update to X and unsetting it after the update, you ensure that only one thread is updating X at a given time.

```fortran
USE omp_lib
INTEGER A(100), X
INTEGER(kind=omp_lock_kind) LCK_X
X=1
CALL omp_init_lock(LCK_X)
!$OMP PARALLEL PRIVATE (I), SHARED (A, X)
!$OMP DO DO I = 3, 100
  A(I) = I * 10
  CALL omp_set_lock(LCK_X)
  X = X + A(I)
  CALL omp_unset_lock(LCK_X)
END DO
!$OMP END DO
!$OMP END PARALLEL
CALL omp_destroy_lock(LCK_X)
END
```

**omp_set_nested(enable_expr)**

Purpose

The **omp_set_nested** subroutine enables or disables nested parallelism.

If **enable_expr** is evaluated to .FALSE., nested parallelism is disabled. Nested parallel regions are serialized, and they are executed by the current thread. This is the default setting.

If **enable_expr** is evaluated to .TRUE., nested parallelism is enabled. Parallel regions that are nested can deploy additional threads to the team. It is up to the runtime environment to determine whether additional threads should be deployed. Therefore, the number of threads used to execute parallel regions may vary from one nested region to the next.

If the routine is called from a portion of the program where the **omp_in_parallel** routine returns true, the routine has no effect.
This subroutine takes precedence over the OMP_NESTED environment variable.

Currently, XL Fortran does not support OpenMP nested parallelism.

**Class**

Subroutine.

**Argument Type and Attributes**

`enable_expr`

Logical.

**Result Type and Attributes**

Default logical.

**Result Value**

None.

```fortran
omp_set_nest_lock(nvar)
```

**Purpose**

The `omp_set_nest_lock` subroutine allows you to set a nestable lock. The thread executing the subroutine will wait until the lock becomes available and then set that lock, incrementing the nesting count. A nestable lock is available if it is owned by the thread executing the subroutine, or is unlocked.

**Class**

Subroutine.

**Argument Type and Attributes**

`nvar` Integer of kind `omp_nest_lock_kind`.

**Result Type and Attributes**

None.

**Result Value**

None.

**Examples**

```fortran
USE omp_lib
INTEGER P
INTEGER A
INTEGER B
INTEGER ( kind=omp_nest_lock_kind ) LCK

CALL omp_init_nest_lock ( LCK )
!$OMP PARALLEL SECTIONS
!$OMP SECTION
CALL omp_set_nest_lock ( LCK )
P = P + A
```
CALL omp_set_nest_lock ( LCK )
P = P + B
CALL omp_unset_nest_lock ( LCK )
CALL omp_unset_nest_lock ( LCK )
!$OMP SECTION
CALL omp_set_nest_lock ( LCK )
P = P + B
CALL omp_unset_nest_lock ( LCK )
!$OMP END PARALLEL SECTIONS
CALL omp_destroy_nest_lock ( LCK )
END

**omp_set_num_threads(number_of_threads_expr)**

**Purpose**

The **omp_set_num_threads** subroutine sets within the runtime environment how many threads to use in the next parallel region. The **number_of_threads_expr** argument is evaluated, and its value is used as the number of threads. If you have enabled dynamic adjustment of the number of threads (see "omp_set_dynamic(enable_expr)" on page 147), **omp_set_num_threads** sets the maximum number of threads to use for the next parallel region. The runtime environment then determines the exact number of threads to use. However, when dynamic adjustment of the number of threads is disabled, **omp_set_num_threads** sets the exact number of threads to use in the next parallel region. If the number of threads you request exceeds the number your execution environment can support, your application will terminate.

This subroutine takes precedence over the **OMP_NUM_THREADS** environment variable.

If you call this subroutine from the dynamic extent of a region executing in parallel, the behavior of the subroutine is undefined.

**Class**

Subroutine.

**Argument Type and Attributes**

**number_of_threads_expr**

integer

**Result Type and Attributes**

None.

**Result Value**

None.
omp_test_lock(svar)

Purpose

The **omp_test_lock** function attempts to set the lock associated with the specified lock variable. It returns **.TRUE.** if it was able to set the lock and **.FALSE.** otherwise. In either case, the calling thread will continue to execute subsequent instructions in the program.

If you call **omp_test_lock** with an uninitialized lock variable, the result of the call is undefined.

Class

Function.

Argument Type and Attributes

svar  Integer of kind **omp_lock_kind**.

Result Type and Attributes

Default logical.

Result Value

**.TRUE.** if the function was able to set the lock. **.FALSE.** otherwise.

Examples

In the following example, a thread repeatedly executes **WORK_A** until it can set the lock variable, **LCK**. When the lock is set, the thread executes **WORK_B**.

```
USE omp_lib
INTEGER LCK
INTEGER ID
CALL omp_init_lock (LCK)
!$OMP PARALLEL SHARED(LCK), PRIVATE(ID)
  ID =omp_get_thread_num()
  DO WHILE (.NOT. omp_test_lock(LCK))
    CALL WORK_A (ID)
  END DO
  CALL WORK_B (ID)
CALL omp_unset_lock (LCK)
!$OMP END PARALLEL
CALL omp_destroy_lock (LCK)
END
```

omp_test_nest_lock(nvar)

Purpose

The **omp_test_nest_lock** subroutine allows you to attempt to set a lock using the same method as **omp_set_nest_lock**, but the execution thread does not wait for confirmation that the lock is available. If the lock is successfully set, the function will increment the nesting count and return the new nesting count. If the lock is unavailable the function returns a value of zero. The result value is always a default integer.
Class

Function.

Argument Type and Attributes

nvar  Integer of kind `omp_nest_lock_kind`.

Result Type and Attributes

Default integer.

Result Value

The new nesting count if the lock is successfully set; otherwise, it returns zero.

**omp_unset_lock(svar)**

Purpose

This subroutine causes the executing thread to release ownership of the specified lock. The lock can then be set by another thread as required. The behavior of the `omp_unset_lock` subroutine is undefined if either of the following conditions occur:

- The calling thread does not own the lock specified.
- The routine is called with an uninitialized lock variable.

Class

Subroutine.

Argument Type and Attributes

svar  Integer of kind `omp_lock_kind`.

Result Type and Attributes

None.

Result Value

None.

Examples

```fortran
USE omp_lib
INTEGER A(100)
INTEGER(kind=omp_lock_kind) LCK_X
CALL omp_init_lock (LCK_X)
!$OMP PARALLEL PRIVATE (I), SHARED (A, X)
!$OMP DO
  DO I = 3, 100
    A(I) = I * 10
    CALL omp_set_lock (LCK_X)
    X = X + A(I)
    CALL omp_unset_lock (LCK_X)
  END DO
```
In this example, the lock variable LCK_X is used to avoid race conditions when updating the shared variable X. By setting the lock before each update to X and unsetting it after the update, you ensure that only one thread is updating X at a given time.

omp_unset_nest_lock(nvar)

Purpose

The omp_unset_nest_lock subroutine allows you to release ownership of a nestable lock. The subroutine decrements the nesting count and releases the associated thread from ownership of the nestable lock.

Class

Subroutine.

Argument Type and Attributes

nvar  Integer of kind omp_lock_kind.

Result Type and Attributes

None.

Result Value

None.

Examples

USE omp_lib
INTEGER P
INTEGER A
INTEGER B
INTEGER ( kind=omp_nest_lock_kind ) LCK

CALL omp_init_nest_lock ( LCK )

!$OMP PARALLEL SECTIONS
!$OMP SECTION
CALL omp_set_nest_lock ( LCK )
P = P + A
CALL omp_set_nest_lock ( LCK )
P = P + B
CALL omp_unset_nest_lock ( LCK )
CALL omp_unset_nest_lock ( LCK )
!$OMP SECTION
CALL omp_set_nest_lock ( LCK )
P = P + B
CALL omp_unset_nest_lock ( LCK )
!$OMP END PARALLEL SECTIONS

CALL omp_destroy_nest_lock ( LCK )
END
Pthreads library module (IBM extension)

The Pthreads Library Module (f_pthread) is a Fortran 90 module that defines data types and routines to make it easier to interface with the AIX pthreads library. The AIX pthreads library is used to parallelize and thread-safe your code.

The f_pthread library module naming convention is the use of the prefix f_ before the corresponding AIX pthreads library routine name or type definition name.

AIX supports both the default POSIX 1003.1-1996 standard, and the Draft 7 POSIX pthreads API. Depending on which invocation command you use, you can compile and link your programs with either the POSIX 1003.1-1996 standard, or the Draft 7 interface libraries. For more information about how to do this, see [Levels of POSIX pthreads API support] [Linking 32-bit] and [Linking 64-bit SMP object files] in the XL Fortran Compiler Reference.

In general, there is a one-to-one corresponding relationship between the procedures in the Fortran 90 module f_pthread and the library routines contained in the AIX pthreads library. However, some of the pthread routines have no corresponding procedures in this module because they are not supported on AIX. One example of these routines is the thread stack address option. There are also some non-pthread interfacing routines contained in the f_pthread library module. The f_maketime routine is one example and is included to return an absolute time in a f_timespec derived type variable.

Most of the routines return an integer value. A return value of 0 will always indicate that the routine call did not result in any error. Any non-zero return value indicates an error. Each error code has a corresponding definition of a system error code in Fortran. These error codes are available as Fortran integer constants. The naming of these error codes in Fortran is consistent with the corresponding AIX error code names. For example, EINVAL is the Fortran constant name of the error code EINVAL on the system. For a complete list of these error codes, refer to the file /usr/include/sys/errno.h.

For more information about the system calls corresponding to the Fortran Pthreads library calls, see the AIX Operating System information.

Pthreads data structures, functions, and subroutines

Pthreads Data Types

- f_pthread_attr_t
- f_pthread_cond_t
- f_pthread_condattr_t
- f_pthread_key_t
- f_pthread_mutex_t
- f_pthread_mutexattr_t
- f_pthread_once_t
- f_pthread_rwlock_t
- f_pthread_rwlockattr_t
- f_pthread_t
- f_sched_param
- f_timespec
Functions that perform operations on thread attribute objects

- `f_pthread_attr_destroy(attr)`
- `f_pthread_attr_getdetachstate(attr, detach)`
- `f_pthread_attr_getguardsize(attr, guardsize)`
- `f_pthread_attr_getinheritsched(attr, inherit)`
- `f_pthread_attr_getschedparam(attr, param)`
- `f_pthread_attr_getschedpolicy(attr, policy)`
- `f_pthread_attr_getscope(attr, scope)`
- `f_pthread_attr_getstackaddr(attr, stackaddr)`
- `f_pthread_attr_getstacksize(attr, ssize)`
- `f_pthread_attr_init(attr)`
- `f_pthread_attr_setdetachstate(attr, detach)`
- `f_pthread_attr_setguardsize(attr, guardsize)`
- `f_pthread_attr_setinheritsched(attr, inherit)`
- `f_pthread_attr_setschedparam(attr, param)`
- `f_pthread_attr_setschedpolicy(attr, policy)`
- `f_pthread_attr_setscope(attr, scope)`
- `f_pthread_attr_setstackaddr(attr, stackaddr)`
- `f_pthread_attr_setstacksize(attr, ssize)`

Functions and Subroutines That Perform Operations on Threads

- `f_pthread_cancel(thread)`
- `f_pthread_cleanup_pop(exec)`
- `f_pthread_cleanup_push(cleanup, flag, arg)`
- `f_pthread_create(thread, attr, flag, ent, arg)`
- `f_pthread_detach(thread)`
- `f_pthread_equal(thread1, thread2)`
- `f_pthread_exit(ret)`
- `f_pthread_getconcurrency()`
- `f_pthread_getschedparam(thread, policy, param)`
- `f_pthread_join(thread, ret)`
- `f_pthread_kill(thread, sig)`
- `f_pthread_self()`
- `f_pthread_setconcurrency(new_level)`
- `f_pthread_setschedparam(thread, policy, param)`

Functions that perform operations on mutex attribute objects

- `f_pthread_mutexattr_destroy(mattr)`
- `f_pthread_mutexattr_getprioceiling(mattr, ceiling)`
- `f_pthread_mutexattr_getprotocol(mattr, proto)`
- `f_pthread_mutexattr_getpshared(mattr, pshared)`
- `f_pthread_mutexattr_gettype(mattr, type)`
- `f_pthread_mutexattr_init(mattr)`
- `f_pthread_mutexattr_setprioceiling(mattr, ceiling)`
- `f_pthread_mutexattr_setprotocol(mattr, proto)`
- `f_pthread_mutexattr_setpshared(mattr, pshared)`
Functions that perform operations on mutex objects
- \( \texttt{f_{-}pthread\_mutex\_attr\_settype(mattr, type)} \)
- \( \texttt{f_{-}pthread\_mutex\_destroy(mutex)} \)
- \( \texttt{f_{-}pthread\_mutex\_getprioceiling(mutex, old)} \)
- \( \texttt{f_{-}pthread\_mutex\_init(mutex, mattr)} \)
- \( \texttt{f_{-}pthread\_mutex\_lock(mutex)} \)
- \( \texttt{f_{-}pthread\_mutex\_setprioceiling(mutex, new, old)} \)
- \( \texttt{f_{-}pthread\_mutex\_trylock(mutex)} \)
- \( \texttt{f_{-}pthread\_mutex\_unlock(mutex)} \)

Functions that perform operations on attribute objects of condition variables
- \( \texttt{f_{-}pthread\_cond\_attr\_destroy(cattr)} \)
- \( \texttt{f_{-}pthread\_cond\_attr\_getpshared(cattr, pshared)} \)
- \( \texttt{f_{-}pthread\_cond\_attr\_init(cattr)} \)
- \( \texttt{f_{-}pthread\_cond\_attr\_setpshared(cattr, pshared)} \)

Functions that perform operations on condition variable objects
- \( \texttt{f_{-}maketime(delay)} \)
- \( \texttt{f_{-}pthread\_cond\_broadcast(cond)} \)
- \( \texttt{f_{-}pthread\_cond\_destroy(cond)} \)
- \( \texttt{f_{-}pthread\_cond\_init(cond, cattr)} \)
- \( \texttt{f_{-}pthread\_cond\_signal(cond)} \)
- \( \texttt{f_{-}pthread\_cond\_timedwait(cond, mutex, timeout)} \)
- \( \texttt{f_{-}pthread\_cond\_wait(cond, mutex)} \)

Functions that perform operations on thread-specific data
- \( \texttt{f_{-}pthread\_getspecific(key, arg)} \)
- \( \texttt{f_{-}pthread\_key\_create(key, dtr)} \)
- \( \texttt{f_{-}pthread\_key\_delete(key)} \)
- \( \texttt{f_{-}pthread\_setspecific(key, arg)} \)

Functions and subroutines that perform operations to control thread cancelability
- \( \texttt{f_{-}pthread\_setcancelstate(state, oldstate)} \)
- \( \texttt{f_{-}pthread\_setcanceltype(type, oldtype)} \)
- \( \texttt{f_{-}pthread\_testcancel()} \)

Functions that perform operations on read-write lock attribute objects
- \( \texttt{f_{-}pthread\_rwlock\_attr\_destroy(rwattr)} \)
- \( \texttt{f_{-}pthread\_rwlock\_attr\_getpshared(rwattr, pshared)} \)
- \( \texttt{f_{-}pthread\_rwlock\_attr\_init(rwattr)} \)
- \( \texttt{f_{-}pthread\_rwlock\_attr\_setpshared(rwattr, pshared)} \)
Functions that perform operations on read-write lock objects

- `f_pthread_rwlock_destroy(rwlock)`
- `f_pthread_rwlock_init(rwlock, rwattr)`
- `f_pthread_rwlock_rdlock(rwlock)`
- `f_pthread_rwlock_tryrdlock(rwlock)`
- `f_pthread_rwlock_trywrlock(rwlock)`
- `f_pthread_rwlock_unlock(rwlock)`
- `f_pthread_rwlock_wrlock(rwlock)`

Functions that perform operations for one-time initialization

- `f_pthread_once(once, initr)`

**f_maketime(delay)**

**Purpose**

This function accepts an integer value specifying a delay in seconds and returns an `f_timespec` type object containing the absolute time, which is `delay` seconds from the calling moment.

**Class**

Function

**Argument Type and Attributes**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>delay</td>
<td>INTEGER(4), INTENT(IN)</td>
<td></td>
</tr>
</tbody>
</table>

**Result Type and Attributes**

<table>
<thead>
<tr>
<th>Result Type</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPE(f_timespec)</td>
<td></td>
</tr>
</tbody>
</table>

**Result Value**

The absolute time, which is `delay` seconds from the calling moment, is returned.

**f_pthread_attr_destroy(attr)**

**Purpose**

This function must be called to destroy any previously initialized thread attribute objects when they will no longer be used. Threads that were created with this attribute object will not be affected in any way by this action. Memory that was allocated when it was initialized will be recollected by the system.

**Class**

Function

**Argument Type and Attributes**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>TYPE(f_pthread_attr_t), INTENT(IN)</td>
<td></td>
</tr>
</tbody>
</table>

**Result Type and Attributes**

<table>
<thead>
<tr>
<th>Result Type</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER(4)</td>
<td></td>
</tr>
</tbody>
</table>
Result Value

On successful completion, this function returns 0. Otherwise, this function returns the following error:

EINVAL
  The argument attr is invalid.

f_pthread_attr_getdetachstate(attr, detach)

Purpose

This function can be used to query the setting of the detach state attribute in the thread attribute object attr. The current setting will be returned through argument detach.

Class

Function

Argument Type and Attributes

attr  TYPE(f_pthread_attr_t), INTENT(IN)
detach INTEGER(4), INTENT(OUT)
  Contains one of the following values:

PTHREAD_CREATE_DETACHED:
  when a thread attribute object of this attribute setting is used to create a new thread, the newly created thread will be in detached state.

PTHREAD_CREATE_UNDETACHED:

PTHREAD_CREATE_JOINABLE:

PTHREAD_CREATE_JOINABLE:
  when a thread attribute object of this attribute setting is used to create a new thread, the newly created thread will be in undetached state. This is the system default.

For more information about these thread states, refer to the AIX Operating System information.

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns the following error:

EINVAL
  The argument attr is invalid.
f_pthread_attr_getguardsize(attr, guardsize)

Purpose

This function is used to get the `guardsize` attribute in the thread attribute object `attr`. The current setting of the attribute will be returned through the argument `guardsize`.

Class

Function

Argument Type and Attributes

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type and Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>TYPE(f_pthread_attr_t), INTENT(IN)</td>
</tr>
<tr>
<td>guardsize</td>
<td>INTEGER(KIND=register_size), INTENT(IN)</td>
</tr>
</tbody>
</table>

where `register_size` is 4 in 32-bit mode, and 8 in 64-bit mode.

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns the following error:

EINVAL
  The argument `attr` is invalid.

f_pthread_attr_getinheritsched(attr, inherit)

Purpose

This function can be used to query the inheritance scheduling attribute in the thread attribute object `attr`. The current setting will be returned through the argument `inherit`.

Class

Function

Argument Type and Attributes

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type and Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>TYPE(f_pthread_attr_t), INTENT(OUT)</td>
</tr>
<tr>
<td>inherit</td>
<td>INTEGER(4)</td>
</tr>
</tbody>
</table>

On return from the function, `inherit` contains one of the following values:

PTHREAD_INHERIT_SCHED:
  indicating that newly created threads will inherit the scheduling property of the parent thread and ignore the scheduling property of the thread attribute object used to create them.
PTHREAD_EXPLICIT_SCHED:
the scheduling property in the thread attribute object will be
assigned to the newly created threads when it is used to create
them.

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns
the following error.

EINVAL
The argument attr is invalid.

ENOSYS
The POSIX priority scheduling option is not implemented on AIX

f_pthread_attr_getschedparam(attr, param)

Purpose

This function can be used to query the scheduling property setting in the thread
attribute object attr. The current setting will be returned in the argument param.
See the AIX system information for a description of the scheduling property
setting.

Class

Function

Argument Type and Attributes

attr  TYPE(f_pthread_attr_t), INTENT(IN)
param TYPE(f_sched_param), INTENT(OUT)

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns
one of the following errors.

EINVAL
The argument attr is invalid.

ENOSYS
The POSIX priority scheduling option is not implemented on AIX.

f_pthread_attr_getschedpolicy(attr, policy)

Purpose

This function can be used to query the scheduling policy attribute setting in the
attribute object attr. The current setting of the scheduling policy will be returned in
the argument policy. The valid scheduling policies on AIX can be found in the AIX Operating System information.

**Class**

Function

**Argument Type and Attributes**

attr TYPE(f_pthread_attr_t), INTENT(IN)
policy INTEGER(4), INTENT(OUT)

**Result Type and Attributes**

INTEGER(4)

**Result Value**

On successful completion, this function returns 0. Otherwise, this function returns one of the following errors.

EINVAL
   The argument attr is invalid.

ENOSYS
   The POSIX priority scheduling option is not implemented on AIX.

**f_pthread_attr_getscope(attr, scope)**

**Purpose**

This function can be used to query the current setting of the scheduling scope attribute in the thread attribute object attr. The current setting will be returned through the argument scope.

**Class**

Function

**Argument Type and Attributes**

attr TYPE(f_pthread_attr_t), INTENT(IN)
scope INTEGER(4), INTENT(OUT)
   On return from the function, scope will contain one of the following values:

PTHREAD_SCOPE_SYSTEM:
   the thread will compete for system resources on a system wide scope.

PTHREAD_SCOPE_PROCESS:
   the thread will compete for system resources locally within the owning process.

**Result Type and Attributes**

INTEGER(4)
Result Value

On successful completion, this function returns 0. Otherwise, this function returns one of the following errors.

EINVAL
  The argument attr is invalid.

ENOSYS
  The POSIX priority scheduling option is not implemented on AIX.

\textbf{f\_pthread\_attr\_getstackaddr}(attr, stackaddr)

Purpose

This function is used to get the stackaddr attribute in the thread attribute object attr. The current setting of the attribute will be returned through the argument stackaddr. The type of the argument stackaddr is Integer pointer. The stackaddr attribute specifies the stack address of a thread created with this attributes object.

Class

Function

Argument Type and Attributes

\begin{itemize}
  \item \textbf{attr} \hspace{1cm} TYPE(f\_pthread\_attr\_t), INTENT(OUT)
  \item \textbf{stackaddr} \hspace{1cm} Integer pointer, INTENT(OUT)
  \item \textbf{ssize} \hspace{1cm} INTEGER(KIND=register\_size), INTENT(OUT)
\end{itemize}

where \textit{register\_size} is 4 in 32-bit mode, and 8 in 64-bit mode.

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns the following error.

EINVAL
  The argument attr is invalid.

\textbf{f\_pthread\_attr\_getstacksize}(attr, ssize)

Purpose

This function can be used to query the current stack size attribute setting in the attribute object attr. If this function executes successfully, the stack size in bytes will be returned in argument ssize.

Class

Function
Argument Type and Attributes

attr    TYPE(f_pthread_attr_t), INTENT(IN)
ssize   INTEGER(KIND=register_size), INTENT(OUT)

where register_size is 4 in 32-bit mode, and 8 in 64-bit mode.

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns one of the following errors.

EINVAL
The argument attr is invalid.

ENOSYS
The POSIX stack size option is not implemented on AIX

f_pthread_attr_init(attr)

Purpose

This function must be called to create and initialize the pthread attribute object attr before it can be used in any way. It will be filled with system default thread attribute values. After it is initialized, certain pthread attributes can be changed and/or set through attribute access procedures. Once initialized, this attribute object can be used to create a thread with the intended attributes. Refer to the AIX Operating System information for descriptions of the default attributes.

Class

Function

Argument Type and Attributes

attr    TYPE(f_pthread_attr_t), INTENT(OUT)

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns one of the following errors.

EINVAL
The argument attr is invalid.

ENOMEM
There is insufficient memory to create this attribute object.
f_pthread_attr_setdetachstate(attr, detach)

Purpose

This function can be used to set the detach state attribute in the thread attribute object attr.

For descriptions of these thread states, refer to the AIX Operating System information.

Class

Function

Argument Type and Attributes

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type and Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>TYPE(f_pthread_attr_t), INTENT(OUT)</td>
</tr>
<tr>
<td>detach</td>
<td>INTEGER(4), INTENT(IN)</td>
</tr>
</tbody>
</table>

Must contain one of the following values:

- **PTHREAD_CREATE_DETACHED**: when a thread attribute object of this attribute setting is used to create a new thread, the newly created thread will be in detached state. This is the system default setting.
- **PTHREAD_CREATE_UNDETACHED**: when a thread attribute object of this attribute setting is used to create a new thread, the newly created thread will be in undetached state.

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns the following error.

EINVAL

The argument attr or detach is invalid.

f_pthread_attr_setguardsize(attr, guardsize)

Purpose

This function is used to set the guardsize attribute in the thread attributes object attr. The new value of this attribute is obtained from the argument guardsize. If guardsize is zero, a guard area will not be provided for threads created with attr. If guardsize is greater than zero, a guard area of at least size guardsize bytes is provided for each thread created with attr.

For a description of guardsize, refer to the AIX Operating System information.

Class

Function
**Argument Type and Attributes**

- **attr**: TYPE(f_pthread_attr_t), INTENT(INOUT)
- **guardsize**: INTEGER(KIND=register_size), INTENT(IN)

Where `register_size` is 4 in 32-bit mode, and 8 in 64-bit mode.

**Result Type and Attributes**

INTEGER(4)

**Result Value**

On successful completion, this function returns 0. Otherwise, this function returns the following error.

- **EINVAL**: The argument **attr** or the argument **guardsize** is invalid.

**f_pthread_attr_setinheritsched(attr, inherit)**

**Purpose**

This function can be used to set the inheritance attribute of the thread scheduling property in the thread attribute object **attr**.

**Class**

Function

**Argument Type and Attributes**

- **attr**: TYPE(f_pthread_attr_t), INTENT(OUT)
- **inherit**: INTEGER(4), INTENT(IN)

Must contain one of the following values:

- **PTHREAD_INHERIT_SCHED**: indicating that newly created threads will inherit the scheduling property of the parent thread and ignore the scheduling property of the thread attribute object used to create them.

- **PTHREAD_EXPLICIT_SCHED**: the scheduling property in the thread attribute object will be assigned to the newly created threads when it is used to create them.

**Result Type and Attributes**

INTEGER(4)

**Result Value**

On successful completion, this function returns 0. Otherwise, this function returns one of the following errors.
EINVAL
The argument attr is invalid.

ENOSYS
The POSIX priority scheduling option is not implemented on AIX.

ENOTSUP
The value of argument inherit is not supported.

f_pthread_attr_setschedparam(attr, param)
Purpose
This function can be used to set the scheduling property attribute in the thread attribute object attr. Threads created with this new attribute object will assume the scheduling property of argument param if they are not inherited from the creating thread. The sched_priority field in argument param indicates the thread’s scheduling priority. The priority field must assume a value in the range of 1-127, where 127 is the most favored scheduling priority while 1 is the least.

Class
Function

Argument Type and Attributes
attr TYPE(f_pthread_attr_t), INTENT(INOUT)
param TYPE(f_sched_param), INTENT(IN)

Result Type and Attributes
INTEGER(4)

Result Value
On successful completion, this function returns 0. Otherwise, this function returns one of the following errors.

EINVAL
The argument attr is invalid.

ENOSYS
The POSIX priority scheduling option is not implemented on AIX.

ENOTSUP
The value of argument param is not supported.

f_pthread_attr_setschedpolicy(attr, policy)
Purpose
After the attribute object is set by this function, threads created with this attribute object will assume the set scheduling policy if the scheduling property is not inherited from the creating thread.

Class
Function


**Argument Type and Attributes**

- **attr**: TYPE(f_thread_attr_t), INTENT(INOUT)
- **policy**: INTEGER(4), INTENT(IN)

  Must contain one of the following values:

  - SCHED_FIFO: indicating a first-in first-out thread scheduling policy.
  - SCHED_RR: indicating a round-robin scheduling policy.
  - SCHED_OTHER: the default scheduling policy.

**Result Type and Attributes**

INTEGER(4)

**Result Value**

On successful completion, this function returns 0. Otherwise, this function returns one of the following errors.

- EINVAL
  - The argument attr is invalid.

- ENOSYS
  - The POSIX priority scheduling option is not implemented on AIX.

- ENOTSUP
  - The value of argument policy is not supported.

**f_pthread_attr_setscope(attr, scope)**

**Purpose**

This function can be used to set the contention scope attribute in the thread attribute object attr.

**Class**

Function

**Argument Type and Attributes**

- **attr**: TYPE(f_pthread_attr_t), INTENT(INOUT)
- **scope**: INTEGER(4), INTENT(IN)

  Must contain one of the following values:

  - PTHREAD_SCOPE_SYSTEM: the thread will compete for system resources on a system-wide scope.
  - PTHREAD_SCOPE_PROCESS: the thread will compete for system resources locally within the owning process.
Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns one of the following errors.

EINVAL
  The argument attr is invalid.

ENOTSUP
  ENOTSUP is returned if the specified scope is PTHREAD_SCOPE_PROCESS.

f_pthread_attr_setstackaddr(attr, stackaddr)

Purpose

This function is used to set the stackaddr attribute in the thread attributes object attr. The new value of this attribute is obtained from the argument stackaddr. The type of the argument stackaddr is Integer pointer. The stackaddr attribute specifies the stack address of a thread created with this attributes object.

Class

Function

Argument Type and Attributes

attr  TYPE(f_pthread_attr_t), INTENT(INOUT)
stackaddr
  Integer pointer, INTENT(IN)

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns the following error.

EINVAL
  The argument attr is invalid.

f_pthread_attr_setstacksize(attr, ssize)

Purpose

This function can be used to set the stack size attribute in the pthread attribute object attr. Argument ssize is an integer indicating the stack size desired in bytes. When a thread is created using this attribute object, the system will allocate a minimum stack size of ssize bytes.
Class

Function

Argument Type and Attributes

attr   TYPE(f_thread_attr_t), INTENT(INOUT)
ssize  INTEGER(KIND=register_size), INTENT(IN)

where register_size is 4 in 32-bit mode, and 8 in 64-bit mode.

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns one of the following errors.

EINVAL
The argument attr or ssize is invalid.

ENOSYS
The POSIX stack size option is not implemented on AIX.

f_thread_attr_t

Purpose

A derived data type whose components are all private. Any object of this type should be manipulated only through the appropriate interfaces provided in this module.

This data type corresponds to the POSIX pthread_attr_t, which is the type of thread attribute object.

Class

Data Type.

f_pthread_cancel(thread)

Purpose

This function can be used to cancel a target thread. How this cancelation request will be processed depends on the state of the cancelability of the target thread. The target thread is identified by argument thread. If the target thread is in deferred-cancel state, this cancelation request will be put on hold until the target thread reaches its next cancelation point. If the target thread disables its cancelability, this request will be put on hold until it is enabled again. If the target thread is in async-cancel state, this request will be acted upon immediately. For further details about thread cancelation and concerns about security, refer to the AIX Operating System information.

Class

Function
Argument Type and Attributes

thread  TYPE(f_pthread_t), INTENT(INOUT)

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns the following error.

EINVAL
    The argument thread is invalid.

f_pthread_cleanup_pop(exec)

Purpose

This subroutine should be paired with f_pthread_cleanup_push in using the cleanup stack for thread safety. If the supplied argument exec contains a non-zero value, the last pushed cleanup function will be popped from the cleanup stack and executed, with the argument arg (from the last f_pthread_cleanup_push) passed to the cleanup function.

If exec contains a zero value, the last pushed cleanup function will be popped from the cleanup stack, but will not be executed.

Class

Subroutine

Argument Type and Attributes

exec  INTEGER(4), INTENT(IN)

Result Type and Attributes

None.

Result Value

None.

f_pthread_cleanup_push(cleanup, flag, arg)

Purpose

This function can be used to register a cleanup subroutine for the calling thread. In case of an unexpected termination of the calling thread, the system will automatically execute the cleanup subroutine in order for the calling thread to terminate safely. The argument cleanup must be a subroutine expecting exactly one argument. If it is executed, the argument arg will be passed to it as the actual argument.
The argument \texttt{arg} is a generic argument that can be of any type and any rank. The actual argument \texttt{arg} must be a variable, and consequently eligible as a left-value in an assignment statement. If you pass an array section with vector subscripts to the argument \texttt{arg}, the result is unpredictable.

If the actual argument \texttt{arg} is an array section, the corresponding dummy argument in subroutine \texttt{cleanup} must be an assumed-shape array. Otherwise, the result is unpredictable.

If the actual argument \texttt{arg} has the pointer attribute that points to an array or array section, the corresponding dummy argument in subroutine \texttt{cleanup} must have a pointer attribute or be an assumed-shape array. Otherwise, the result is unpredictable.

For a normal execution path, this function must be paired with a call to \texttt{f_pthread_cleanup_pop}.

The argument \texttt{flag} must be used to convey the property of argument \texttt{arg} exactly to the system.

\textbf{Class}

Function

\textbf{Argument Type and Attributes}

\texttt{cleanup}
\begin{quote}
A subroutine that has one dummy argument.
\end{quote}

\texttt{flag}
\begin{quote}
Flag is an INTEGER(4), INTENT(IN) argument that can contain one of, or a combination of, the following constants:
\begin{description}
\item[FLAG_CHARACTER:] if the entry subroutine \texttt{cleanup} expects an argument of type CHARACTER in any way or any form, this flag value must be included to indicate this fact. However, if the subroutine expects a Fortran 90 pointer pointing to an argument of type CHARACTER, the \texttt{FLAG_DEFAULT} value should be included instead.
\item[FLAG_ASSUMED_SHAPE:] if the entry subroutine \texttt{cleanup} has a dummy argument that is an assumed-shape array of any rank, this flag value must be included to indicate this fact.
\item[FLAG_DEFAULT:] otherwise, this flag value is needed.
\end{description}
\end{quote}

\texttt{arg}
\begin{quote}
A generic argument that can be of any type, kind, and rank.
\end{quote}

\textbf{Result Type and Attributes}

\texttt{INTEGER(4)}

\textbf{Result Value}

On successful completion, this function returns 0. Otherwise, this function returns one of the following errors.
ENOMEM
   The system cannot allocate memory to push this routine.

EAGAIN
   The system cannot allocate resources to push this routine.

EINVAL
   The argument flag is invalid.

**f_pthread_cond_broadcast(cond)**

**Purpose**

This function will unblock all threads waiting on the condition variable cond. If there is no thread waiting on this condition variable, the function will still succeed, but the next caller to f_pthread_cond_wait will be blocked, and will wait on the condition variable cond.

**Class**

Function

**Argument Type and Attributes**

cond   TYPE(f_pthread_cond_t), INTENT(INOUT)

**Result Type and Attributes**

INTEGER(4)

**Result Value**

On successful completion, this function returns 0. Otherwise, this function returns following error.

EINVAL
   The argument cond is invalid.

**f_pthread_cond_destroy(cond)**

**Purpose**

This function can be used to destroy those condition variables that are no longer required. The target condition variable is identified by the argument cond. System resources allocated during initialization will be recollected by the system. For further details about thread synchronization and condition variable usage, refer to the AIX Operating System information.

**Class**

Function

**Argument Type and Attributes**

cond   TYPE(f_pthread_cond_t), INTENT(INOUT)

**Result Type and Attributes**

INTEGER(4)
Result Value

On successful completion, this function returns 0. Otherwise, this function returns one of the following errors.

EBUSY
The condition variable cond is being used by another thread.

EINVAL
The argument cond is invalid.

f_pthread_cond_init(cond, cattr)

Purpose

This function can be used to dynamically initialize a condition variable cond. Its attributes will be set according to the attribute object cattr, if it is provided; otherwise, its attributes will be set to the system default. After the condition variable is initialized successfully, it can be used to synchronize threads. For further details about thread synchronization and condition variable usage, refer to the AIX Operating System information.

Another method of initializing a condition variable is to initialize it statically using the Fortran constant PTHREAD_COND_INITIALIZER.

Class
Function

Argument Type and Attributes

cond TYPE(f_pthread_cond_t), INTENT(INOUT)
cattr TYPE(f_pthread_condattr_t), INTENT(IN), OPTIONAL

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns one of the following errors.

EBUSY
The condition variable is already in use. It is initialized and not destroyed.

EINVAL
The argument cond or cattr is invalid.

f_pthread_cond_signal(cond)

Purpose

This function will unblock at least one thread waiting on the condition variable cond. If there is no thread waiting on this condition variable, the function will still succeed, but the next caller to f_pthread_cond_wait will be blocked, and will wait on the condition variable cond. For further details about thread synchronization and condition variable usage, refer to the AIX Operating System information.
Class  
Function  

Argument Type and Attributes  
cond  TYPE(f_pthread_cond_t), INTENT(INOUT)  

Result Type and Attributes  
INTEGER(4)  

Result Value  
On successful completion, this function returns 0. Otherwise, this function returns the following error.  
EINVAL  
The argument cond is invalid.  

f_pthread_cond_t  
Purpose  
A derived data type whose components are all private. Any object of this type should be manipulated through the appropriate interfaces provided in this module. In addition, objects of this type can be initialized at compile time using the Fortran constant PTHREAD_COND_INITIALIZER.  

This data type corresponds to the POSIX pthread_cond_t, which is the type of condition variable object.  

Class  
Data Type.  

f_pthread_cond_timedwait(cond, mutex, timeout)  
Purpose  
This function can be used to wait for a certain condition to occur. The argument mutex must be locked before calling this function. The mutex is unlocked atomically and the calling thread waits for the condition to occur. The argument timeout specifies a deadline before which the condition must occur. If the deadline is reached before the condition occurs, the function will return an error code. This function provides a cancelation point in that the calling thread can be canceled if it is in the enabled state.  

The argument timeout will specify an absolute date of the form: Oct. 31 10:00:53, 1998. For related information, see f_maketime and f_timespec. For a description of the absolute date, refer to the AIX Operating System information.  

Class  
Function
**Argument Type and Attributes**

- **cond** TYPE(f pthread_cond_t), INTENT(INOUT)
- **mutex** TYPE(f pthread_mutex_t), INTENT(INOUT)
- **timeout** TYPE(f timespec), INTENT(IN)

**Result Type and Attributes**

INTEGER(4)

**Result Value**

On successful completion, this function returns 0. Otherwise this function returns one of the following errors:

- **EINVAL**
  - The argument cond, mutex, or timeout is invalid.

- **EDEADLK**
  - The argument mutex is not locked by the calling thread.

- **ETIMEDOUT**
  - The waiting deadline was reached before the condition occurred.

**f pthread cond wait(cond, mutex)**

**Purpose**

This function can be used to wait for a certain condition to occur. The argument mutex must be locked before calling this function. The mutex is unlocked atomically, and the calling thread waits for the condition to occur. If the condition does not occur, the function will wait until the calling thread is terminated in another way. This function provides a cancelation point in that the calling thread can be canceled if it is in the enabled state.

**Class**

Function

**Argument Type and Attributes**

- **cond** TYPE(f pthread_cond_t), INTENT(INOUT)
- **mutex** TYPE(f pthread_mutex_t), INTENT(INOUT)

**Result Type and Attributes**

INTEGER(4)

**Result Value**

When this function executes successfully, the mutex is locked again before the function returns. Otherwise, this function returns one of the following errors:

- **EINVAL**
  - The argument cond or mutex is invalid.
EDEADLK
The mutex is not locked by the calling thread.

f_pthread_condattr_destroy(cattr)

Purpose
This function can be called to destroy the condition variable attribute objects that are no longer required. The target object is identified by the argument cattr. The system resources allocated when it is initialized will be recollected.

Class
Function

Argument Type and Attributes

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type and Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>cattr</td>
<td>TYPE(f_pthread_condattr_t), INTENT(INOUT)</td>
</tr>
</tbody>
</table>

Result Type and Attributes

INTEGER(4)

Result Value
On successful completion, this function returns 0. Otherwise, this function returns the following error.

EINVAL
The argument cattr is invalid.

f_pthread_condattr_getpshared(cattr, pshared)

Purpose
This function can be used to query the process-shared attribute of the condition variable attributes object identified by the argument cattr. The current setting of this attribute will be returned in the argument pshared.

Class
Function

Argument Type and Attributes

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type and Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>cattr</td>
<td>TYPE(f_pthread_condattr_t), INTENT(IN)</td>
</tr>
<tr>
<td>pshared</td>
<td>INTEGER(4), INTENT(OUT)</td>
</tr>
</tbody>
</table>

On successful completion, pshared contains one of the following values:

PTHREAD_PROCESS_SHARED
The condition variable can be used by any thread that has access to the memory where it is allocated, even if these threads belong to different processes.

PTHREAD_PROCESS_PRIVATE
The condition variable shall only be used by threads within the same process as the thread that created it.
Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns the following error.

EINVAL
   The argument cattr is invalid.

\textbf{f\_pthread\_condattr\_init(cattr)}

\textbf{Purpose}

Use this function to initialize a condition variable attributes object cattr with the default value for all of the attributes defined by the implementation. Attempting to initialize an already initialized condition variable attributes object results in undefined behavior. After a condition variable attributes object has been used to initialize one or more condition variables, any function affecting the attributes object (including destruction) does not affect any previously initialized condition variables.

\textbf{Class}

Function

\textbf{Argument Type and Attributes}

\begin{itemize}
\item \texttt{cattr} \hspace{1cm} TYPE(f\_pthread\_condattr\_t), INTENT(OUT)
\end{itemize}

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns the following error.

ENOMEM
   There is insufficient memory to initialize the condition variable attributes object.

\textbf{f\_pthread\_condattr\_setpshared(cattr, pshared)}

\textbf{Purpose}

This function is used to set the process-shared attribute of the condition variable attributes object identified by the argument cattr. Its process-shared attribute will be set according to the argument pshared.

\textbf{Class}

Function
Argument Type and Attributes

cattr TYPE(f_pthread_condattr_t), INTENT(INOUT)
pshared is an INTEGER(4), INTENT(IN) argument that must contain one of the following values:

PTHREAD_PROCESS_SHARED
   Specifies that the condition variable can be used by any thread that has access to the memory where it is allocated, even if these threads belong to different processes.

PTHREAD_PROCESS_PRIVATE
   Specifies that the condition variable shall only be used by threads within the same process as the thread that created it. This is the default setting of the attribute.

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns the following error.

EINVAL
   The value specified by the argument cattr or pshared is invalid.

f_pthread_condattr_t

Purpose

A derived data type whose components are all private. Any object of this type should be manipulated only through the appropriate interfaces provided in this module.

This data type corresponds to the POSIX pthread_condattr_t, which is the type of condition variable attribute object.

Class

Data Type

f_pthread_create(thread, attr, flag, ent, arg)

Purpose

This function is used to create a new thread in the current process. The newly created thread will assume the attributes defined in the thread attribute object attr, if it is provided. Otherwise, the new thread will have system default attributes. The new thread will begin execution at the subroutine ent, which is required to have one dummy argument. The system will pass the argument arg to the thread entry subroutine ent as its actual argument. The argument flag is used to inform the system of the property of the argument arg. When the execution returns from the entry subroutine ent, the new thread will terminate automatically.
If subroutine \texttt{ent} was declared such that an explicit interface would be required if it was called directly, then an explicit interface is also required when it is passed as an argument to this function.

The argument \texttt{arg} is a generic argument that can be of any type and any rank. The actual argument \texttt{arg} must be a variable, and consequently eligible as a left-value in an assignment statement. If you pass an array section with vector subscripts to the argument \texttt{arg}, the result is unpredictable.

If the actual argument \texttt{arg} is an array section, the corresponding dummy argument in subroutine \texttt{ent} must be an assumed-shape array. Otherwise, the result is unpredictable.

If the actual argument \texttt{arg} has the pointer attribute that points to an array or array section, the corresponding dummy argument in subroutine \texttt{ent} must have a pointer attribute or be an assumed-shape array. Otherwise, the result is unpredictable.

\textbf{Class}

Function

\textbf{Argument Type and Attributes}

\begin{description}
  \item[\texttt{thread}] TYPE(f\_pthread\_t), INTENT(OUT)
    \begin{itemize}
      \item On successful completion of the function, \texttt{f\_pthread\_create} stores the ID of the created thread in the \texttt{thread}.
    \end{itemize}
  \item[\texttt{attr}] TYPE(f\_pthread\_attr\_t), INTENT(IN)
  \item[\texttt{flag}] INTEGER(4), INTENT(IN)
    \begin{itemize}
      \item The argument \texttt{flag} must convey the property of the argument \texttt{arg} exactly to the system. The argument \texttt{flag} can be one of, or a combination of, the following constants:
        \begin{itemize}
          \item [\texttt{FLAG\_CHARACTER}]
            \begin{itemize}
              \item if the entry subroutine \texttt{ent} expects an argument of type \texttt{CHARACTER} in any way or any form, this flag value must be included to indicate this fact. However, if the subroutine expects a Fortran 90 pointer pointing to an argument of type \texttt{CHARACTER}, the \texttt{FLAG\_DEFAULT} value should be included instead.
            \end{itemize}
          \item [\texttt{FLAG\_ASSUMED\_SHAPE}]
            \begin{itemize}
              \item if the entry subroutine \texttt{ent} has a dummy argument which is an assumed-shape array of any rank, this flag value must be included to indicate this fact.
            \end{itemize}
          \item [\texttt{FLAG\_DEFAULT}]
            \begin{itemize}
              \item otherwise, this flag value is needed.
            \end{itemize}
        \end{itemize}
  \end{itemize}
  \item[\texttt{ent}] A subroutine that has one dummy argument.
  \item[\texttt{arg}] A generic argument that can be of any type, kind, and rank.
\end{description}

\textbf{Result Type and Attributes}

INTEGER(4)
**Result Value**

On successful completion, this function returns 0. Otherwise, this function returns one of the following errors.

**EAGAIN**

The system does not have enough resources to create a new thread.

**EINVAL**

The argument `thread`, `attr`, or `flag` is invalid.

**ENOMEM**

The system does not have sufficient memory to create a new thread.

**f_pthread_detach(thread)**

**Purpose**

This function is used to indicate to the pthreads library implementation that storage for the thread whose thread ID is specified by the argument `thread` can be claimed when this thread terminates. If the thread has not yet terminated, `f_pthread_detach` shall not cause it to terminate. Multiple `f_pthread_detach` calls on the same target thread cause an error.

**Class**

Function

**Argument Type and Attributes**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread</td>
<td>TYPE(f_pthread_t), INTENT(IN)</td>
<td></td>
</tr>
</tbody>
</table>

**Result Type and Attributes**

INTEGER(4)

**Result Value**

On successful completion, this function returns 0. Otherwise, this function returns the following error.

**EINVAL**

The argument `thread` is invalid.

**f_pthread_equal(thread1, thread2)**

**Purpose**

This function can be used to compare whether two thread ID’s identify the same thread or not.

**Class**

Function

**Argument Type and Attributes**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread1</td>
<td>TYPE(f_pthread_t), INTENT(IN)</td>
<td></td>
</tr>
</tbody>
</table>
thread2
    TYPE(f_pthread_t), INTENT(IN)

Result Type and Attributes

LOGICAL(4)

Result Value

TRUE  The two thread ID’s identify the same thread.
FALSE The two thread ID’s do not identify the same thread.

f_pthread_exit(ret)

Purpose

This subroutine can be called explicitly to terminate the calling thread before it returns from the entry subroutine. The actions taken depend on the state of the calling thread. If it is in non-detached state, the calling thread will wait to be joined. If the thread is in detached state, or when it is joined by another thread, the calling thread will terminate safely. First, the cleanup stack will be popped and executed, and then any thread-specific data will be destructed by the destructors. Finally, the thread resources are freed and the argument ret will be returned to the joining threads. The argument ret of this subroutine is optional. Currently, argument ret is limited to be an Integer pointer. If it is not an Integer pointer, the behavior is undefined. Calling f_pthread_exit will not automatically free all of the memory allocated to a thread. To avoid memory leaks, finalization must be handled separately from f_pthread_exit.

This subroutine never returns. If argument ret is not provided, NULL will be provided as this thread’s exit status.

Class

Subroutine

Argument Type and Attributes

ret   [Integer pointer] OPTIONAL, INTENT(IN)

Result Type and Attributes

None

Result Value

None

f_ppthread_getconcurrency() Purpose

This function returns the value of the concurrency level set by a previous call to the f_ppthread_setconcurrency function. If the f_ppthread_setconcurrency function was not previously called, this function returns zero to indicate that the system is maintaining the concurrency level.

Chapter 7. Parallel programming with XL Fortran   181
For a description of the concurrency level, refer to the AIX Operating System information.

Class
Function

Argument Type and Attributes
None

Result Type and Attributes

INTEGER(4)

Result Value
This function returns the value of the concurrency level set by a previous call to the f_pthread_setconcurrency function. If the f_pthread_setconcurrency function was not previously called, this function returns 0.

\[ \text{f_pthread_getschedparam}(\text{thread}, \text{policy}, \text{param}) \]

Purpose
This function can be used to query the current setting of the scheduling property of the target thread. The target thread is identified by argument \text{thread}. Its scheduling policy will be returned through argument \text{policy} and its scheduling property through argument \text{param}. The sched_priority field in \text{param} defines the scheduling priority. The priority field will assume a value in the range of 1-127, where 127 is the most favored scheduling priority while 1 is the least.

Class
Function

Argument Type and Attributes
\text{thread} \quad \text{TYPE(f_pthread_t), INTENT(IN)}
\text{policy} \quad \text{INTEGER(4), INTENT(OUT)}
\text{param} \quad \text{TYPE(f_sched_param), INTENT(OUT)}

Result Type and Attributes

INTEGER(4)

Result Value
On successful completion, this function returns 0. Otherwise, this function returns one of the following errors.

\text{ENOSYS}
   The POSIX priority scheduling option is not implemented on AIX.

\text{ESRCH}
   The target thread does not exist.
f_pthread_getspecific(key, arg)

Purpose

This function can be used to retrieve the thread-specific data associated with key. Note that the argument arg is not optional in this function as it will return the thread-specific data. After execution of the procedure, the argument arg holds a pointer to the data, or NULL if there is no data to retrieve. The argument arg must be an Integer pointer, or the result is undefined.

The actual argument arg must be a variable, and consequently eligible as a left-value in an assignment statement. If you pass an array section with vector subscripts to the argument arg, the result is unpredictable.

Class

Function

Argument Type and Attributes

key TYPE(f_pthread_key_t), INTENT(IN)
arg [Integer pointer] INTENT(OUT)

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns the following error.

EINVAL
  The argument key is invalid.

f_pthread_join(thread, ret)

Purpose

This function can be called to join a particular thread designated by the argument thread. If the target thread is in non-detached state and is already terminated, this call will return immediately with the target thread’s status returned in argument ret if it is provided. The argument ret is optional. Currently, ret must be an Integer pointer if it is provided.

If the target thread is in detached state, it is an error to join it.

Class

Function

Argument Type and Attributes

thread TYPE(f_pthread_t), INTENT(IN)
ret [Integer pointer] INTENT(OUT), OPTIONAL
Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns one of the following errors.

EDEADLK
   This call will cause a deadlock, or the calling thread is trying to join itself.

EINVAL
   The argument thread is invalid.

ESRCH
   The argument thread designates a thread which does not exist or is in detached state.

f_pthread_key_create(key, dtr)

Purpose

This function can be used to acquire a thread-specific data key. The key will be returned in the argument key. The argument dtr is a subroutine that will be used to destruct the thread-specific data associated with this key when any thread terminates after this calling point. The destructor will receive the thread-specific data as its argument. The destructor itself is optional. If it is not provided, the system will not invoke any destructor on the thread-specific data associated with this key. Note that the number of thread-specific data keys is limited in each process. It is the user’s responsibility to manage the usage of the keys. The per-process limit can be checked by the Fortran constant PTHREAD_DATAKEYS_MAX.

Class

Function

Argument Type and Attributes

key TYPE(f_pthread_key_t), INTENT(OUT)
dtr External, optional subroutine

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns one of the following errors.

EAGAIN
   The maximum number of keys has been exceeded.

EINVAL
   The argument key is invalid.
There is insufficient memory to create this key.

**f_pthread_key_delete(key)**

**Purpose**

This function will destroy the thread-specific data key identified by the argument *key*. It is the user’s responsibility to ensure that there is no thread-specific data associated with this key. This function does not call any destructor on the thread’s behalf. After the key is destroyed, it can be reused by the system for **f_pthread_key_create** requests.

**Class**

Function

**Argument Type and Attributes**

$key$: TYPE(f_pthread_key_t), INTENT(INOUT)

**Result Type and Attributes**

INTEGER(4)

**Result Value**

On successful completion, this function returns 0. Otherwise, this function returns one of the following errors.

- **EINVAL**: The argument *key* is invalid.
- **EBUSY**: There is still data associated with this key.

**f_pthread_key_t**

**Purpose**

A derived data type whose components are all private. Any object of this type should be manipulated only through the appropriate interfaces provided in this module.

This data type corresponds to the POSIX **pthread_key_t**, which is the type of key object for accessing thread-specific data.

**Class**

Data Type

**f_pthread_kill(thread, sig)**

**Purpose**

This function can be used to send a signal to a target thread. The target thread is identified by argument *thread*. The signal which will be sent to the target thread is identified in argument *sig*. If *sig* contains value zero, error checking will be done by the system but no signal will be sent. For further details about signal...
management in multi-threaded systems, refer to the AIX Operating System information.

Class

Function

**Argument Type and Attributes**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread</td>
<td>TYPE(f_thread_t),</td>
<td>INTENT(INOUT)</td>
</tr>
<tr>
<td>sig</td>
<td>INTEGER(4),</td>
<td>INTENT(IN)</td>
</tr>
</tbody>
</table>

**Result Type and Attributes**

INTEGER(4)

**Result Value**

On successful completion, this function returns 0. Otherwise, this function returns one of the following errors.

**EINVAL**

The argument `thread` or `sig` is invalid.

**ESRCH**

The target thread does not exist.

**f_pthread_mutex_destroy(mutex)**

**Purpose**

This function should be called to destroy those mutex objects that are no longer required. In this way, the system can recollect the memory resources. The target mutex object is identified by the argument `mutex`.

Class

Function

**Argument Type and Attributes**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>mutex</td>
<td>TYPE(f_pthread_mutex_t),</td>
<td>INTENT(INOUT)</td>
</tr>
</tbody>
</table>

**Result Type and Attributes**

INTEGER(4)

**Result Value**

On successful completion, this function returns 0. Otherwise, this function returns one of the following errors.

**EBUSY**

The target mutex is locked or referenced by another thread.

**EINVAL**

The argument `mutex` is invalid.
f_pthread_mutex_getprioceiling(mutex, old)

Purpose

This function can be used to dynamically query the priority ceiling attribute of the mutex object identified by the argument mutex. The current ceiling value will be returned through the argument old.

Class

Function

Argument Type and Attributes

mutex TYPE(f_pthread_mutex_t), INTENT(IN)
old INTEGER(4), INTENT(OUT)

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns one of the following errors.

ENOSYS
   This function is not implemented on AIX.

f_pthread_mutex_init(mutex, mattr)

Purpose

This function can be used to initialize the mutex object identified by argument mutex. The initialized mutex will assume attributes set in the mutex attribute object mattr, if it is provided. If mattr is not provided, the system will initialize the mutex to have default attributes. After it is initialized, the mutex object can be used to synchronize accesses to critical data or code. It can also be used to build more complicated thread synchronization objects.

Another method to initialize mutex objects is to statically initialize them through the Fortran constant PTHREAD_MUTEX_INITIALIZER. If this method of initialization is used it is not necessary to call the function before using the mutex objects.

Class

Function

Argument Type and Attributes

mutex TYPE(f_pthread_mutex_t), INTENT(OUT)
mattr TYPE(f_pthread_mutexattr_t), INTENT(IN), OPTIONAL

Result Type and Attributes

INTEGER(4)
Result Value

On successful completion, this function returns 0. Otherwise, this function returns one of the following errors.

EAGAIN
The system did not have enough resources to initialize this mutex.

EBUSY
This mutex is already in use. It was initialized and not destroyed.

EINVAL
The argument mutex or mattr is invalid.

ENOMEM
There is insufficient memory to initialize this mutex.

f_pthread_mutex_lock(mutex)

Purpose

This function can be used to acquire ownership of the mutex object. (In other words, the function will lock the mutex.) If the mutex has already been locked by another thread, the caller will wait until the mutex is unlocked. If the mutex is already locked by the caller itself, an error will be returned to prevent recursive locking.

Class

Function

Argument Type and Attributes

mutex  TYPE(f_pthread_mutex_t), INTENT(INOUT)

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns one of the following errors.

EDEADLK
The mutex is locked by the calling thread already.

EINVAL
The argument mutex is invalid.

f_pthread_mutex_setprioceiling(mutex, new, old)

Purpose

This function can be used to dynamically set the priority ceiling attribute of the mutex object identified by the argument mutex. The new ceiling will be set to the value contained in the argument new. The previous ceiling will be returned through the argument old. The argument new should assume an integer value with a range from 1 to 127.
Class
Function

Argument Type and Attributes
mutex  TYPE(f_pthread_mutex_t), INTENT(INOUT)
new    INTEGER(4), INTENT(IN)
old    INTEGER(4), INTENT(OUT)

Result Type and Attributes
INTEGER(4)

Result Value
On successful completion, this function returns 0. Otherwise, this function returns
the following error.
ENOSYS
This function is not implemented on AIX.

f_pthread_mutex_t
Purpose
A derived data type whose components are all private. Any object of this type
should be manipulated through the appropriate interfaces provided in this module.
In addition, objects of this type can be initialized statically through the Fortran
constant PTHREAD_MUTEX_INITIALIZER.

This data type corresponds to the POSIX pthread_mutex_t, which is the type of
mutex object.

Class
Data Type

f_pthread_mutex_trylock(mutex)
Purpose
This function can be used to acquire ownership of the mutex object. (In other
words, the function will lock the mutex.) If the mutex has already been locked by
another thread, the function returns the error code EBUSY. The calling thread can
check the return code to take further actions. If the mutex is already locked by the
caller itself, an error will be returned to prevent recursive locking.

Class
Function

Argument Type and Attributes
mutex  TYPE(f_pthread_mutex_t), INTENT(INOUT)
Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns one of the following errors.

EBUSY
   The target mutex is locked or referenced by another thread.

EDEADLK
   The mutex is locked by the calling thread already.

EINVAL
   The argument mutex is invalid.

f_pthread_mutex_unlock(mutex)

Purpose

This function releases the mutex object’s ownership in order to allow other threads to lock the mutex.

Class

Function

Argument Type and Attributes

mutex   TYPE(f_pthread_mutex_t), INTENT(INOUT)

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns one of the following errors.

EINVAL
   The argument mutex is invalid.

EPERM
   The mutex is not locked by the calling thread.

f_pthread_mutexattr_destroy(mattr)

Purpose

This function can be used to destroy a mutex attribute object that has been initialized previously. Allocated memory will then be recollected. A mutex created with this attribute will not be affected by this action.

Class

Function
Argument Type and Attributes

\texttt{mattr} \hspace{1em} \text{TYPE(f pthread_mutexattr_t), INTENT(INOUT)}

Result Type and Attributes

\text{INTEGER(4)}

Result Value

On successful completion, this function returns 0. Otherwise, this function returns the following error.

\text{EINVAL}

The argument \texttt{mattr} is invalid.

\texttt{f pthread_mutexattr_getprioceiling(mattr, ceiling)}

Purpose

This function can be used to query the mutex priority ceiling attribute in the mutex attribute object identified by argument \texttt{mattr}. The ceiling attribute will be returned through argument \texttt{ceiling}.

Class

Function

Argument Type and Attributes

\texttt{mattr} \hspace{1em} \text{TYPE(f pthread_mutexattr_t), INTENT(IN)}
\texttt{ceiling} \hspace{1em} \text{INTEGER(4), INTENT(OUT)}

Result Type and Attributes

\text{INTEGER(4)}

Result Value

On successful completion, this function returns 0. Otherwise, this function returns the following error.

\text{ENOSYS}

This function is not implemented on AIX.

\texttt{f pthread_mutexattr_getprotocol(mattr, proto)}

Purpose

This function can be used to query the current setting of mutex protocol attribute in the mutex attribute object identified by argument \texttt{mattr}. The protocol attribute will be returned through argument \texttt{proto}.

Class

Function
Argument Type and Attributes

mattr  TYPE(f_thread_mutexattr_t), INTENT(IN)
proto  INTEGER(4), INTENT(OUT)

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns the following error.

ENOSYS
This function is not implemented on AIX.

f_pthread_mutexattr_getpshared(mattr, pshared)

Purpose

This function is used to query the process-shared attribute in the mutex attributes object identified by the argument mattr. The current setting of the attribute will be returned through the argument pshared.

Class

Function

Argument Type and Attributes

mattr  TYPE(f_thread_mutexattr_t), INTENT(IN)
pshared INTEGER(4), INTENT(IN)

On return from this function, pshared contains one of the following values:

PTHREAD_PROCESS_SHARED
The mutex can be operated upon by any thread that has access to the memory where the mutex is allocated, even if the mutex is allocated in memory that is shared by multiple processes.

PTHREAD_PROCESS_PRIVATE
The mutex will only be operated upon by threads created within the same process as the thread that initialized the mutex.

Result Type and Attributes

INTEGER(4)

Result Value

If this function completes successfully, value 0 is returned and the value of the process-shared attribute is returned through the argument pshared. Otherwise, the following error will be returned:

EINVAL
The argument mattr is invalid.
f_pthread_mutexattr_gettype(mattr, type)

Purpose

This function is used to query the mutex type attribute in the mutex attributes object identified by the argument mattr.

If this function completes successfully, value 0 is returned and the type attribute will be returned through the argument type.

Class

Function

Argument Type and Attributes

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>mattr</td>
<td>TYPE(f_pthread_mutexattr_t), INTENT(IN)</td>
<td></td>
</tr>
<tr>
<td>type</td>
<td>INTEGER(4), INTENT(OUT)</td>
<td></td>
</tr>
</tbody>
</table>

On return from this function, type contains one of the following values:

- **PTHREAD_MUTEX_NORMAL**
  - This type of mutex does not detect deadlock. A thread attempting to relock this mutex without first unlocking it will deadlock. Attempting to unlock a mutex locked by a different thread results in undefined behavior.

- **PTHREAD_MUTEX_ERRORCHECK**
  - This type of mutex provides error checking. A thread attempting to relock this mutex without first unlocking it will return with an error. A thread attempting to unlock a mutex which another thread has locked will return an error. A thread attempting to unlock an unlocked mutex will return with an error.

- **PTHREAD_MUTEX_RECURSIVE**
  - A thread attempting to relock this mutex without first unlocking it will succeed in locking the mutex. The relocking deadlock that can occur with mutexes of type **PTHREAD_MUTEX_NORMAL** cannot occur with this type of mutex. Multiple locks of this mutex require the same number of unlocks to release the mutex before another thread can acquire the mutex.

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns the following error.

- **EINVAL**
  - The argument is invalid.
f_pthread_mutexattr_init(mattr)

Purpose

This function can be used to initialize a mutex attribute object before it can be used in any other way. The mutex attribute object will be returned through argument mattr.

Class

Function

Argument Type and Attributes

mattr  TYPE(f_pthread_mutexattr_t), INTENT(OUT)

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns the following error.

EINVAL
  The argument mattr is invalid.
ENOMEM
  There is insufficient memory to create the object.

f_pthread_mutexattr_setprioceiling(mattr, ceiling)

Purpose

This function can be used to set the mutex priority ceiling attribute in the mutex attribute object identified by the argument mattr. Argument ceiling is an integer with a range from 1 to 127. This attribute has an effect only if the mutex priority protection protocol is used.

Class

Function

Argument Type and Attributes

mattr  TYPE(f_pthread_mutexattr_t), INTENT(INOUT)

ceiling  INTEGER(4), INTENT(IN)

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns the following error.
f_pthread_mutexattr_setprotocol(mattr, proto)

Purpose

This function can be used to set the mutex protocol attribute in the mutex attribute object identified by argument mattr. Argument proto identifies the mutex protocol to be set. For descriptions of the set of valid values for proto, refer to the AIX Operating System information.

Class

Function

Argument Type and Attributes

mattr TYPE(f_pthread_mutexattr_t), INTENT(INOUT)
proto INTEGER(4), INTENT(IN)

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns the following error.

ENOSYS
This function is not implemented on AIX.

f_pthread_mutexattr_setpshared(mattr, pshared)

Purpose

This function is used to set the process-shared attribute of the mutex attributes object identified by the argument mattr.

Class

Function

Argument Type and Attributes

mattr TYPE(f_pthread_mutexattr_t), INTENT(INOUT)
pshared INTEGER(4), INTENT(IN)

Must contain one of the following values:

PTHREAD_PROCESS_SHARED
Specifies the mutex can be operated upon by any thread that has access to the memory where the mutex is allocated, even if the mutex is allocated in memory that is shared by multiple processes.

PTHREAD_PROCESS_PRIVATE
Specifies the mutex will only be operated upon by threads created
within the same process as the thread that initialized the mutex. This is the default setting of the attribute.

**Result Type and Attributes**

INTEGER(4)

**Result Value**

On successful completion, this function returns 0. Otherwise, this function returns the following error.

**EINVAL**

The argument is invalid.

### f_pthread_mutexattr_settype(mattr, type)

**Purpose**

This function is used to set the mutex type attribute in the mutex attributes object identified by the argument *mattr*. The argument type identifies the mutex type attribute to be set.

For further details about the type of a mutex, refer to the AIX Operating System information.

**Class**

Function

**Argument Type and Attributes**

- **mattr** TYPE(f_pthread_mutexattr_t), INTENT(INOUT)
- **type** INTEGER(4), INTENT(IN)

Must contain one of the following values:

**PTHREAD_MUTEX_NORMAL**

This type of mutex does not detect deadlock. A thread attempting to relock this mutex without first unlocking it will deadlock. Attempting to unlock a mutex locked by a different thread results in undefined behavior.

**PTHREAD_MUTEX_ERRORCHECK**

This type of mutex provides error checking. A thread attempting to relock this mutex without first unlocking it will return with an error. A thread attempting to unlock a mutex which another thread has locked will return an error. A thread attempting to unlock an unlocked mutex will return with an error.

**PTHREAD_MUTEX_RECURSIVE**

A thread attempting to relock this mutex without first unlocking it will succeed in locking the mutex. The relocking deadlock that can occur with mutexes of type **PTHREAD_MUTEX_NORMAL** cannot occur with this type of mutex. Multiple locks of this mutex require the same number of unlocks to release the mutex before another thread can acquire the mutex.
PTHREAD_MUTEX_DEFAULT
The same as PTHREAD_MUTEX_NORMAL.

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns the following error.

EINVAL
One of the arguments is invalid.

f_pthread_mutexattr_t

Purpose

A derived data type whose components are all private. Any object of this type should be manipulated only through the appropriate interfaces provided in this module.

This data type corresponds to the POSIX pthread_mutexattr_t, which is the type of mutex attribute object.

Class

Data Type

f_pthread_once(once, initr)

Purpose

This function can be used to initialize those data required to be initialized only once. The first thread calling this function will call initr to do the initialization. Other threads calling this function afterwards will have no effect. Argument initr must be a subroutine without dummy arguments.

Class

Function

Argument Type and Attributes

once TYPE(f_pthread_once_t), INTENT(INOUT)
initr A subroutine that has no dummy arguments.

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns the following error.
EINVAL

The argument once or initr is invalid.

f_pthread_once_t

Purpose

A derived data type whose components are all private. Any object of this type should be manipulated through the appropriate interfaces provided in this module. However, objects of this type can only be initialized through the Fortran constant PTHREAD_ONCE_INIT.

Class

Data Type

f_pthread_rwlock_destroy(rwlock)

Purpose

This function destroys the read-write lock object specified by the argument rwlock and releases any resources used by the lock.

Class

Function

Argument Type and Attributes

rwlock

TYPE(f_pthread_rwlock_t), INTENT(INOUT)

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns one of the following errors.

EBUSY

The target read-write object is locked.

EINVAL

The argument rwlock is invalid.

f_pthread_rwlock_init(rwlock, rwattr)

Purpose

This function initializes the read-write lock object specified by rwlock with the attribute specified by the argument rwattr. If the optional argument rwattr is not provided, the system will initialize the read-write lock object with the default attributes. After it is initialized, the lock can be used to synchronize access to critical data. With a read-write lock, many threads can have simultaneous read-only access to data, while only one thread can have write access at any given
time and no other readers or writers are allowed. For further details of the thread synchronization and read-write lock usage, refer to the AIX Operating System information.

Another method to initialize read-write lock objects is to statically initialize them through the Fortran constant `PTHREAD_RWLOCK_INITIALIZER`. If this method of initialization is used, it is not necessary to call this function before using the read-write lock objects.

Class

Function

Argument Type and Attributes

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type and Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rwlock</code></td>
<td><code>TYPE(f_pthread_rwlock_t), INTENT(OUT)</code></td>
</tr>
<tr>
<td><code>rwattr</code></td>
<td><code>TYPE(f_pthread_rwlockattr_t), INTENT(IN), OPTIONAL</code></td>
</tr>
</tbody>
</table>

Result Type and Attributes

`INTEGER(4)`

Result Value

On successful completion, this function returns 0. Otherwise, this function returns one of the following errors.

- **EAGAIN**
  - The system did not have enough resources to initialize this read-write lock.

- **ENOMEM**
  - There is insufficient memory to initialize this read-write lock.

- **EBUSY**
  - This read-write lock is already in use. It was initialized and not yet destroyed.

- **EINVAL**
  - The argument `rwlock` or `rwattr` is invalid.

- **EPERM**
  - The caller does not have privilege to perform the operation.

### f_pthread_rwlock_rdlock(rwlock)

**Purpose**

This function applies a read lock to the read-write lock specified by the argument `rwlock`. The calling thread acquires the read lock if a writer does not hold the lock and there are no writes blocked on the lock. Otherwise, the calling thread will not acquire the read lock. If the read lock is not acquired, the calling thread blocks (that is, it does not return from the `f_pthread_rwlock_rdlock` call) until it can acquire the lock. Results are undefined if the calling thread holds a write lock on `rwlock` at the time the call is made. A thread may hold multiple concurrent read locks on `rwlock` (that is, successfully call the `f_pthread_rwlock_rdlock` function `n` times). If so, the thread must perform matching unlocks (that is, it must call the `f_pthread_rwlock_unlock` function `n` times).
<table>
<thead>
<tr>
<th>Class</th>
<th>Function</th>
</tr>
</thead>
</table>

**Argument Type and Attributes**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type and Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>rwlock</td>
<td>TYPE(f_pthread_rwlock_t), INTENT(INOUT)</td>
</tr>
</tbody>
</table>

**Result Type and Attributes**

<table>
<thead>
<tr>
<th>Result</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER(4)</td>
<td></td>
</tr>
</tbody>
</table>

**Result Value**

On successful completion, this function returns 0. Otherwise, this function returns one of the following errors.

- **EAGAIN**
  The read-write lock could not be acquired because the maximum number of read locks for rwlock has been exceeded.

- **EINVAL**
  The argument rwlock does not refer to an initialized read-write lock object.

**f_pthread_rwlock_t**

**Purpose**

A derived data type whose components are all private. Any object of this type should be manipulated only through the appropriate interfaces provided in this module. In addition, objects of this type can be initialized statically through the Fortran constant `PTHREAD_RWLOCK_INITIALIZER`.

This data type corresponds to the AIX data type `pthread_rwlock_t`, which is the type of the read-write lock objects.

**Class**

Data Type

**f_pthread_rwlock_tryrdlock(rwlock)**

**Purpose**

This function applies a read lock like the `f_pthread_rwlock_rdlock` function with the exception that the function fails if any thread holds a write lock on rwlock or there are writers blocked on rwlock. In that case, the function returns EBUSY. The calling thread can check the return code to take further actions.

**Class**

Function

**Argument Type and Attributes**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type and Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>rwlock</td>
<td>TYPE(f_pthread_rwlock_t), INTENT(INOUT)</td>
</tr>
</tbody>
</table>
**Result Type and Attributes**

INTEGER(4)

**Result Value**

This function returns zero if the lock for reading on the read-write lock object specified by *rwlock* is acquired. Otherwise, one of the following errors will be returned:

**EAGAIN**
The read-write lock could not be acquired because the maximum number of read locks for *rwlock* has been exceeded

**EBUSY**
The read-write lock could not be acquired for reading because a writer holds the lock or was blocked on it.

**EDEADLK**
The current thread already owns the read-write lock for writing.

**EINVAL**
The argument *rwlock* does not refer to an initialized read-write lock object.

**f_pthread_rwlock_trywrlock(rwlock)**

**Purpose**

This function applies a write lock like the *f_pthread_rwlock_wrlock* function with the exception that the function fails if any thread currently holds *rwlock* (for reading or writing). In that case, the function returns **EBUSY**. The calling thread can check the return code to take further actions.

**Class**

Function

**Argument Type and Attributes**

*rwlock*

```fortran
TYPE(f_pthread_rwlock_t), INTENT(INOUT)
```

**Result Type and Attributes**

INTEGER(4)

**Result Value**

This function returns zero if the lock for writing on the read-write lock object specified by *rwlock* is acquired. Otherwise, one of the following errors will be returned:

**EBUSY**
The read-write lock could not be acquired for reading because a writer holds the lock or was blocked on it.

**EDEADLK**
The current thread already owns the read-write lock for writing.
EINVAL
The argument rwlock does not refer to an initialized read-write lock object.

f_pthread_rwlock_unlock(rwlock)
Purpose
This function is used to release a lock held on the read-write lock object specified by the argument rwlock. If this function is called to release a read lock from the read-write lock object and there are other read locks currently held on this read-write lock object, the read-write lock object remains in the read locked state. If this function releases the calling thread’s last read lock on this read-write lock object, then the calling thread is no longer one of the owners of the object. If this function releases the last read lock for this read-write lock object, the read-write lock object will be put in the unlocked state with no owners.

Class
Function

Argument Type and Attributes
rwlock
   TYPE(f_pthread_rwlock_t), INTENT(INOUT)

Result Type and Attributes
INTEGER(4)

Result Value
On successful completion, this function returns 0. Otherwise, this function returns one of the following errors.
EINVAL
The argument rwlock does not refer to an initialized read-write lock object.
EPERM
The current thread does not own the read-write lock.

f_pthread_rwlock_wrlock(rwlock)
Purpose
This function applies a write lock to the read-write lock specified by the argument rwlock. The calling thread acquires the write lock if no other thread (reader or writer) holds the read-write lock rwlock. Otherwise, the thread blocks (that is, does not return from the f_pthread_rwlock_wrlock call) until it acquires the lock. Results are undefined if the calling thread holds the read-write lock (whether a read or write lock) at the time the call is made.

Class
Function

Argument Type and Attributes
rwlock
   TYPE(f_pthread_rwlock_t), INTENT(INOUT)
Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns
the following error.

EINVAL

The argument rwlock does not refer to an initialized read-write lock object.

f_pthread_rwlockattr_destroy(rwattr)

Purpose

This function destroys a read-write lock attributes object specified by the argument
rwattr which has been initialized previously. A read-write lock created with this
attribute will not be affected by the action.

Class

Function

Argument Type and Attributes

rwattr  TYPE(f_pthread_rwlockattr_t), INTENT(INOUT)

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns
the following error.

EINVAL

The argument rwattr is invalid.

f_pthread_rwlockattr_getpshared(rwattr, pshared)

Purpose

This function is used to obtain the value of the process-shared attribute from the
initialized read-write lock attributes object specified by the argument rwattr. The
current setting of this attribute will be returned in the argument pshared.

Class

Function

Argument Type and Attributes

rwattr  TYPE(f_pthread_rwlockattr_t), INTENT(IN)

pshared

INTEGER(4), INTENT(OUT)
On return from this function, the value of `pshared` will be one of the following:

**PTHREAD_PROCESS_SHARED**

The read-write lock can be operated upon by any thread that has access to the memory where it is allocated, even if these threads belong to different processes.

**PTHREAD_PROCESS_PRIVATE**

The read-write lock shall only be used by threads within the same process as the thread that created it.

### Result Type and Attributes

**INTEGER(4)**

### Result Value

If this function completes successfully, value 0 is returned and the value of the process-shared attribute of `rwattr` is stored into the object specified by the argument `pshared`. Otherwise, the following error will be returned:

**EINVAL**

The argument `rwattr` is invalid.

### `f_pthread_rwlockattr_init(rwattr)`

**Purpose**

This function initializes a read-write lock attributes object specified by `rwattr` with the default value for all of the attributes.

**Class**

Function

### Argument Type and Attributes

- **rwattr**
  
  TYPE(f_pthread_rwlockattr_t), INTENT(OUT)

### Result Type and Attributes

**INTEGER(4)**

### Result Value

On successful completion, this function returns 0. Otherwise, this function returns the following error:

**ENOMEM**

There is insufficient memory to initialize the read-write lock attributes object.
f_pthread_rwlockattr_setpshared(rwattr, pshared)

Purpose

This function is used to set the process-shared attribute in an initialized read-write lock attributes object specified by the argument rwattr, based on the value provided by the argument pshared.

Class

Function

Argument Type and Attributes

rwattr  TYPE(f_pthread_rwlockattr_t), INTENT(INOUT)

pshared  INTEGER(4), INTENT(IN)

Must be one of the following:

PTHREAD_PROCESS_SHARED
  Specifies the read-write lock can be operated upon by any thread that has access to the memory where it is allocated, even if these threads belong to different processes.

PTHREAD_PROCESS_PRIVATE
  Specifies the read-write lock shall only be used by threads within the same process as the thread that created it. This is the default setting of the attribute.

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns the following error:

EINVAL
  The argument rwattr is invalid.

f_pthread_rwlockattr_t

Purpose

This is a derived data type whose components are all private. Any object of this type should be manipulated only through the appropriate interfaces provided in this module.

This data type corresponds to the data type pthread_rwlockattr_t, which is the type of the read-write lock attributes objects.

Class

Data Type
**f_pthread_self()**

**Purpose**

This function can be used to return the thread ID of the calling thread.

**Class**

Function

**Argument Type and Attributes**

None

**Result Type and Attributes**

*TYPE(f_pthread_t)*

**Result Value**

The calling thread’s ID is returned.

**f_pthread_setcancelstate(state, oldstate)**

**Purpose**

This function can be used to set the thread’s cancelability state. The new state will be set according to the argument *state*. The old state will be returned in the argument *oldstate*.

**Class**

Function

**Argument Type and Attributes**

- **state**  
  INTEGER(4), INTENT(IN)
  Must contain one of the following:
  
  **PTHREAD_CANCEL_DISABLE**:  
  The thread’s cancelability is disabled.
  
  **PTHREAD_CANCEL_ENABLE**:  
  The thread’s cancelability is enabled.

- **oldstate**  
  INTEGER(4), INTENT(OUT)
  On return from this function, *oldstate* will contain one of the following values:
  
  **PTHREAD_CANCEL_DISABLE**:  
  The thread’s cancelability is disabled.
  
  **PTHREAD_CANCEL_ENABLE**:  
  The thread’s cancelability is enabled.

**Result Type and Attributes**

INTEGER(4)
Result Value

On successful completion, this function returns 0. Otherwise, this function returns the following error.

EINVAL

The argument state is invalid.

f_pthread_setcanceltype(type, oldtype)

Purpose

This function can be used to set the thread’s cancelability type. The new type will be set according to the argument type. The old type will be returned in argument oldtype.

Class

Function

Argument Type and Attributes

type INTEGER(4), INTENT(IN)

Must contain one of the following values:

PTHREAD_CANCEL_DEFERRED:
Cancellation request will be delayed until a cancelation point.

PTHREAD_CANCELASYNCHRONOUS:
Cancellation request will be acted upon immediately. This may cause unexpected results.

oldtype

INTEGER(4), INTENT(OUT)

On return from this procedure, oldtype will contain one of the following values:

PTHREAD_CANCEL_DEFERRED:
Cancellation request will be delayed until a cancelation point.

PTHREAD_CANCELASYNCHRONOUS:
Cancellation request will be acted upon immediately. This may cause unexpected results.

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns the following error.

EINVAL

The argument type is invalid.
**f_pthread_setconcurrency(new_level)**

**Purpose**

This function is used to inform the pthreads library implementation of desired concurrency level as specified by the argument *new_level*. The actual level of concurrency provided by the implementation as a result of this function call is unspecified. For further details about the concurrency level, refer to the AIX Operating System information.

**Class**

Function

**Argument Type and Attributes**

- **new_level**
  - INTEGER(4), INTENT(IN)

**Result Type and Attributes**

- INTEGER(4)

**Result Value**

On successful completion, this function returns 0. Otherwise, this function returns one of the following errors.

- **EAGAIN**
  - The value specified by *new_level* would cause system resource to be exceeded.

- **EINVAL**
  - The value specified by *new_level* is negative.

**f_pthread_setschedparam(thread, policy, param)**

**Purpose**

This function can be used to dynamically set the scheduling policy and the scheduling property of a thread. The target thread is identified by argument *thread*. The new scheduling policy for the target thread is provided through argument *policy*. The valid scheduling policies on AIX can be found in the AIX Operating System information. The new scheduling property of the target thread will be set to the value provided by argument *param*. The sched_priority field in *param* defines the scheduling priority. Its range is 1-127.

The new policy cannot be set to first-in first-out or round-robin unless the caller has root authority. For more details about when the new scheduling property has effect on the target thread, refer to the AIX Operating System information.

**Class**

Function

**Argument Type and Attributes**

- **thread**
  - TYPE(f_pthread_t), INTENT(INOUT)
policy INTEGER(4), INTENT(IN)
param TYPE(f_sched_param), INTENT(IN)

Result Type and Attributes
INTEGER(4)

Result Value
On successful completion, this function returns 0. Otherwise, this function returns one of the following errors

EINVAL
The argument thread or param is invalid.
ENOSYS
The POSIX priority scheduling option is not implemented on AIX.
ENOTSUP
The value of argument policy or param is not supported.
EPERM
The target thread is not permitted to perform the operation or is in a mutex protocol already.
ESRCH
The target thread does not exist or is invalid.

f_pthread_setspecific(key, arg)
Purpose
This function can be used to set the calling thread’s specific data associated with the key identified by argument key. The argument arg, which is optional, identifies the thread-specific data to be set. If arg is not provided, the thread-specific data will be set to NULL, which is the initial value for each thread. Only an Integer pointer can be passed as the arg argument. If arg is not an Integer pointer, the result is undefined.

The actual argument arg must be a variable, and consequently eligible as a left-value in an assignment statement. If you pass an array section with vector subscripts to the argument arg, the result is unpredictable.

Class
Function

Argument Type and Attributes
key TYPE(f_pthread_key_t), INTENT(IN)
arg INTEGER pointer, INTENT(IN), OPTIONAL

Result Type and Attributes
INTEGER(4)
Result Value

On successful completion, this function returns 0. Otherwise, this function returns one of the following errors

EINVAL
  The argument key is invalid.

ENOMEM
  There is insufficient memory to associate the data with the key.

f_pthread_t
Purpose

A derived data type whose components are all private. Any object of this type should be manipulated only through the appropriate interfaces provided in this module.

This data type corresponds to the POSIX pthread_t, which is the type of thread object.

Class
Data Type

f_pthread_testcancel()
Purpose

This subroutine provides a cancelation point in a thread. When it is called, any pending cancelation request will be acted upon immediately if it is in the enabled state.

Class
Subroutine

Argument Type and Attributes
None

Result Type and Attributes
None

f_sched_param
Purpose

This data type corresponds to the AIX system data structure sched_param, which is a system data type. See AIX Operating System information for further details.

This is a public data structure defined as:

type f_sched_param
  sequence
    integer sched_priority
end type f_sched_param
**Class**

Data Type

**f_sched_yield()**

**Purpose**

This function is used to force the calling thread to relinquish the processor until it again becomes the head of its thread list.

**Class**

Function

**Argument Type and Attributes**

None.

**Result Type and Attributes**

INTEGER(4)

**Result Value**

If this function completes successfully, value 0 is returned. Otherwise, a value of -1 will be returned.

**f_timespec**

**Purpose**

This is a Fortran definition of the AIX system data structure timespec. Within the Fortran Pthreads module, objects of this type are used to specify an absolute date and time. This *deadline absolute date* is used when waiting on a POSIX condition variable.

In 32–bit mode, *f_timespec* is defined as:

```
TYPE F_Timespec
  SEQUENCE
  INTEGER(4) tv_sec
  INTEGER(KIND=REGISTER_SIZE) tv_nsec
END TYPE F_Timespec
```

In 64–bit mode, *f_timespec* is defined as:

```
TYPE F_Timespec
  SEQUENCE
  INTEGER(4) tv_sec
  INTEGER(4) pad
  INTEGER(KIND=REGISTER_SIZE) tv_nsec
END TYPE F_Timespec
```

See AIX Operating System information for further details.

**Class**

Data Type
Chapter 8. Interlanguage calls

Your Fortran application can perform interlanguage calls to routines written in a language other than Fortran.

The guidelines assume 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.

The rules are:

- 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:

  ```fortran
  @process mixed
  external c_func  ! With MIXED, we can call C_FUNC, not just c_func.
  integer aBC, ABC ! With MIXED, these are different variables.
  common /xYz/ aBc ! The same applies to the common block names.
  common /XYZ/ ABC ! XYZ and XYZ are external names that are visible during linking.
  end
  ```

- Names for module procedures are formed by concatenating `__` (two underscores), the module name, `_IMOD_` (for intrinsic modules) or `_NMOD_` (for non-intrinsic modules), and the name of the module procedure. For example, module procedure `MYPROC` in module `MYMOD` has the external name `__mymod_NMOD_Myproc`.

  **Note:** Symbolic debuggers and other tools should account for this naming scheme when debugging XL Fortran programs that contain module procedures. For example, some debuggers default to lowercase for program and symbolic names. This behavior should be changed to use mixed case when debugging XL Fortran programs with module procedures.

- The XL compilers generate code that uses `main` as an external entry point name. You can only use `main` as an external name in these contexts:
A Fortran program or local-variable name. (This restriction means that you cannot use main as a binding label, or for the name of an external function, external subroutine, block data program unit, or common block. References to such an object use the compiler-generated main instead of your own.)

- The name of the top-level main function in a C program.

Some other potential naming conflicts may occur when linking a program. For instructions on avoiding them, see [Linking new objects with existing ones] and [Avoiding naming conflicts during linking] in the XL Fortran Compiler Reference.

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. Or you can use the -brename linker option on AIX to rename the symbol:

```
xlf90 -brename:old_name,new_name interlanguage_calls.f
```

---

### Mixed-language input and output

To improve performance, the XL Fortran runtime 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.

Mixing code compiled by multiple Fortran compilers, for example xlf and g77, could face similar problems. The safest approach is to treat the code compiled by another Fortran compiler as non-Fortran code. 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 runtime 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 runtime option to disable buffering for I/O operations.

For more information on the flush and getfd procedures, see the [Service and utility procedures] topic in the XL Fortran Language Reference. For more information on the buffering runtime option, see [Setting runtime options] in the XL Fortran Compiler Reference.
Mixing Fortran and C++

When mixing Fortran and C++ in the same program, you need to invoke the C++ compiler to correctly link the final program.

Most of the information in this section applies to Fortran and 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 your C++ compiler’s invocation command, like `xlc` or `g++`, 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](in the XL Fortran Compiler Reference).

```fortran
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 7. Main Fortran program that calls C++ (main1.f)

```c
#include <stdio.h>

extern "C" 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;
}

extern "C" 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 8. 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 << end1; }};

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 << end1; }};

Figure 9. 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, or make use of the standards-compliant VALUE attribute. For example:
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 221 and %VAL and %REF in the XL Fortran Language Reference for more details.

### Passing data from one language to another

You need to account for corresponding data types in Fortran and C when passing data from one language to another.

The Corresponding data types in Fortran and C table shows the data types available in the XL Fortran and C languages. Further topics detail how Fortran arguments can be passed by reference to C programs. To use the Fortran 2003 Standard interoperability features, see the BIND attribute and ISO_C_BINDING module in the XL Fortran Language Reference.

### Passing arguments between languages

When calling Fortran procedures, the C routines must pass arguments as pointers to the types listed in the following table.

**Table 19. Corresponding data types in Fortran and C**

<table>
<thead>
<tr>
<th>XL Fortran Data Types</th>
<th>XL C/C++ Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER(1), BYTE</td>
<td>signed char</td>
</tr>
<tr>
<td>INTEGER(2)</td>
<td>signed short</td>
</tr>
<tr>
<td>INTEGER(4)</td>
<td>signed int</td>
</tr>
<tr>
<td>INTEGER(8)</td>
<td>signed long long</td>
</tr>
<tr>
<td>REAL, REAL(4)</td>
<td>float</td>
</tr>
<tr>
<td>REAL(8), DOUBLE PRECISION</td>
<td>double</td>
</tr>
<tr>
<td>REAL(16)</td>
<td>long double (see note 1)</td>
</tr>
<tr>
<td>COMPLEX, COMPLEX(4)</td>
<td>float _Complex</td>
</tr>
<tr>
<td>COMPLEX(8), DOUBLE COMPLEX</td>
<td>double _Complex</td>
</tr>
<tr>
<td>COMPLEX(16)</td>
<td>long double _Complex (see note 1)</td>
</tr>
<tr>
<td>LOGICAL(1)</td>
<td>unsigned char</td>
</tr>
<tr>
<td>LOGICAL(2)</td>
<td>unsigned short</td>
</tr>
<tr>
<td>LOGICAL(4)</td>
<td>unsigned int</td>
</tr>
<tr>
<td>LOGICAL(8)</td>
<td>unsigned long long</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>char</td>
</tr>
<tr>
<td>CHARACTER(n)</td>
<td>char[n]</td>
</tr>
<tr>
<td>Integer POINTER</td>
<td>void *</td>
</tr>
<tr>
<td>Array</td>
<td>array</td>
</tr>
<tr>
<td>Sequence-derived type</td>
<td>structure (with C/C++ -qalign=packed option)</td>
</tr>
</tbody>
</table>

**Note:**
1. Requires C/C++ compiler -qlongdbl option.
Note:

1. In interlanguage communication, it is often necessary to use the %VAL built-in function, or the standards-compliant VALUE attribute, and the %REF built-in function that are defined in “Passing arguments by reference or by value” on page 221.

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) by value to a C function that you have not declared with an explicit interface.

3. The Fortran-derived type and the C structure must match in the number, data type, and length of subobjects to be compatible data types.

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 Standard interoperability features provided by XL Fortran, see the Language interoperability features section in the XL Fortran Language Reference.

**Passing global variables between languages**

To access a C data structure from within a Fortran program or to access a common block from within a C program, follow these steps:

1. Create a named common block that provides a one-to-one mapping of the C structure members. If you have an unnamed common block, change it to a named one. Name the common block with the name of the C structure.

2. Declare the C structure as a global variable by putting its declaration outside any function or inside a function with the extern qualifier.

3. Compile the C source file with -qalign=packed.

```fortran
program cstruct
    real(8) a,d
    integer b,c
    .
    .
    common /mystuff/ a,b,c,d
    .
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 or put #pragma into the struct.

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 runtime 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 the Directives section in the XL Fortran Language Reference, and Chapter 12, “Sample Fortran programs,” on page 283.

Common blocks that are declared THREADPRIVATE can be accessed using a C global variable that is declared as THREADPRIVATE.
Passing character types between languages

One difficult aspect of interlanguage calls is passing character strings between languages. The difficulty is due to the following underlying differences in the way that different languages represent such entities:

- The only character type in Fortran is **CHARACTER**, which is stored as a set of contiguous bytes, one character per byte. The length is not stored as part of the entity. Instead, it is passed by value as an extra argument at the end of the declared argument list when the entity is passed as an argument. The size of the argument is 4 or 8 bytes, depending on the compilation mode used (32- or 64-bit, respectively).

- 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 (described in the XL Fortran Compiler Reference).

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:

```fortran
! Initialize a character string to pass to C.
character*6 message1 /'Hello\0'/
! Initialize a character string as usual, and append the null later.
character*5 message2 />'world'/
! Pass both strings to a C function that takes 2 (char *) arguments.
call cfunc(%ref(message1), %ref(message2 // '\0'))
end
```

For compatibility with C language usage, you can encode the following escape sequences in XL Fortran character strings:

*Table 20. Escape sequences for character strings*

<table>
<thead>
<tr>
<th>Escape</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\b</td>
<td>Backspace</td>
</tr>
<tr>
<td>\f</td>
<td>Form feed</td>
</tr>
<tr>
<td>\n</td>
<td>New-line</td>
</tr>
<tr>
<td>\t</td>
<td>Tab</td>
</tr>
<tr>
<td>\0</td>
<td>Null</td>
</tr>
<tr>
<td>'</td>
<td>Apostrophe (does not terminate a string)</td>
</tr>
<tr>
<td>&quot;</td>
<td>Double quotation mark (does not terminate a string)</td>
</tr>
<tr>
<td>\ \</td>
<td>Backslash</td>
</tr>
<tr>
<td>\x</td>
<td>(x), where (x) is any other character (the backslash is ignored)</td>
</tr>
</tbody>
</table>

If you do not want the backslash interpreted as an escape character within strings, you can compile with the `-qnoescape` option.
Passing arrays between languages

Fortran stores array elements in ascending storage units in column-major order. C stores array elements in row-major order. Fortran array indexes start at 1, while C array indexes start at 0.

The following example shows how a two-dimensional array that is declared by A(3,2) is stored in Fortran and C.

<table>
<thead>
<tr>
<th>Fortran Element Name</th>
<th>C Element Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lowest storage unit</td>
<td></td>
</tr>
<tr>
<td>A(1,1)</td>
<td>A[0][0]</td>
</tr>
<tr>
<td>A(2,1)</td>
<td>A[0][1]</td>
</tr>
<tr>
<td>A(3,1)</td>
<td>A[1][0]</td>
</tr>
<tr>
<td>A(1,2)</td>
<td>A[1][1]</td>
</tr>
<tr>
<td>A(2,2)</td>
<td>A[2][0]</td>
</tr>
<tr>
<td>Highest storage unit</td>
<td></td>
</tr>
<tr>
<td>A(3,2)</td>
<td>A[2][1]</td>
</tr>
</tbody>
</table>

Table 21. Corresponding array layouts for Fortran and C. The Fortran array reference A(X,Y,Z) can be expressed in C as a[Z-1][Y-1][X-1]. Keep in mind that although C passes individual scalar array elements by value, it passes arrays by reference.

To pass all or part of a Fortran array to another language, you can use Fortran 90/Fortran 95 array notation:

```fortran
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:

```fortran
INTERFACE
FUNCTION CFUNC (ARRAY, PTR1, PTR2)
    INTEGER, DIMENSION (:), DIMENSION (:), DIMENSION (:), DIMENSION (:), DIMENSION (:)
    :: ARRAY
    INTEGER, DIMENSION (:), DIMENSION (:), DIMENSION (:)
    :: PTR1
    REAL, POINTER :: PTR2
    ! Change this : to *.
    ! Change this : to *
    ! and remove the POINTER
    ! attribute.
    ! Remove this POINTER
    ! attribute or change to TARGET.
END FUNCTION
END INTERFACE
```
Passing pointers between languages

Integer POINTERs always represent the address of the pointee object and must be passed by value:

```
CALL CFUNC(%VAL(INTPTR))
```

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 POINTERs can also be passed back and forth between languages but only if there is no explicit interface for the called procedure or if the argument in the explicit interface does not have a POINTER attribute or assumed-shape declarator. You can remove any POINTER attribute or change it to TARGET and can change any deferred-shape array declarator to be explicit-shape or assumed-size.

Because of XL Fortran’s call-by-reference conventions, you must pass even scalar values from another language as the address of the value, rather than the value itself. For example, a C function passing an integer value x to Fortran must pass &x. Also, a C function passing a pointer value p to Fortran so that Fortran can use it as an integer POINTER must declare it as void **p. A C array is an exception: you can pass it to Fortran without the & operator.

Passing arguments by reference or by value

To call subprograms written in languages other than Fortran (for example, user-written C programs, or operating system routines), the actual arguments may need to be passed by a method different from the default method used by Fortran. C routines, including those in system libraries, such as libc.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 built-in function or VALUE attribute and the %REF built-in function 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 32-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(I)`. If the actual argument is a `CHARACTER(I)`, the compiler pads it on the left with zeros to a 32-bit value, regardless of whether you specified the `-qctyplss` compiler option.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>In 64-bit mode</strong></td>
<td></td>
</tr>
<tr>
<td>If the actual argument is one of the following:</td>
<td></td>
</tr>
<tr>
<td>• An integer or a logical that is shorter than 64 bits, it is sign-extended to a 64-bit value.</td>
<td></td>
</tr>
<tr>
<td>• Of type real or complex, it is passed as multiple 64-bit intermediate values.</td>
<td></td>
</tr>
<tr>
<td>• Of sequence-derived type, it is passed as multiple 64-bit intermediate values.</td>
<td></td>
</tr>
<tr>
<td>Byte-named constants and variables are passed as if they were <code>INTEGER(I)</code>. If the actual argument is a <code>CHARACTER(I)</code>, the compiler pads it on the left with zeros to a 64-bit value, regardless of whether you specified the <code>-qctyplss</code> compiler option.</td>
<td></td>
</tr>
</tbody>
</table>

If you specified the `-qautodbl` compiler option, any padded storage space is not passed except for objects of derived type.

**VALUE attribute**

Specifies an argument association between a dummy and an actual argument that allows you to pass the dummy argument with the value of the actual argument. Changes to the value or definition status of the dummy argument do not affect the actual argument.

You must specify the `VALUE` attribute for dummy arguments only.

You must not use the `%VAL` or `%REF` built-in functions to reference a dummy argument with the `VALUE` attribute, or the associated actual argument.

A referenced procedure that has a dummy argument with the `VALUE` attribute must have an explicit interface.

You must not specify the `VALUE` attribute with the following:
- Arrays
- Derived types with `ALLOCATABLE` components
- Dummy procedures

```fortran
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

Example with VALUE attribute
Program validexml
    integer :: x = 10, y = 20
    print *, 'before calling: ', x, y
    call intersub(x, y)
    print *, 'after calling: ', x, y
contains
    subroutine intersub(x,y)
        integer, value :: x
        integer y
        x = x + y
        y = x*y
        print *, 'in subroutine after changing: ', x, y
    end subroutine
    end program validexml

Expected output:
    before calling: 10 20
    in subroutine after changing: 30 600
    after calling: 10 600

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
    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
"Order of arguments in argument list" on page 232

Type encoding and checking
Runtime 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, Volumes 1 - 6 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), general-purpose registers (GPRs), vector registers (VPRs) and minimizing the saving and restoring of registers on subroutine entry and exit. The linkage convention allows for argument passing and return values to be in FPRs, GPRs, or both.

The following table lists floating-point registers and their functions. The floating-point registers are double precision (64 bits).

Table 22. Floating-point register usage across calls

<table>
<thead>
<tr>
<th>Register</th>
<th>Preserved Across Calls</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>no</td>
<td>FP parameter 1, function return 1.</td>
</tr>
<tr>
<td>2</td>
<td>no</td>
<td>FP parameter 2, function return 2.</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>13</td>
<td>no</td>
<td>FP parameter 13, function return 13.</td>
</tr>
<tr>
<td>14-31</td>
<td>yes</td>
<td></td>
</tr>
</tbody>
</table>

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

Table 23. General-purpose register usage across calls

<table>
<thead>
<tr>
<th>Register</th>
<th>Preserved Across Calls</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>yes</td>
<td>Stack pointer.</td>
</tr>
<tr>
<td>2</td>
<td>yes</td>
<td>TOC pointer.</td>
</tr>
<tr>
<td>3</td>
<td>no</td>
<td>1st word of arg list; return value 1.</td>
</tr>
<tr>
<td>4</td>
<td>no</td>
<td>2nd word of arg list; return value 2.</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>10</td>
<td>no</td>
<td>8th word of arg list; return value 8.</td>
</tr>
<tr>
<td>11</td>
<td>no</td>
<td>DSA pointer to internal procedure (Env).</td>
</tr>
<tr>
<td>12</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>13-31</td>
<td>yes</td>
<td></td>
</tr>
</tbody>
</table>
Table 23. General-purpose register usage across calls (continued)

<table>
<thead>
<tr>
<th>Register</th>
<th>Preserved Across Calls</th>
<th>Use</th>
</tr>
</thead>
</table>
| If a register is not designated as preserved, its contents may be changed during the call, and the caller is responsible for saving any registers whose values are needed later. Conversely, if a register is supposed to be preserved, the callee is responsible for preserving its contents across the call, and the caller does not need any special action.

The following table lists special-purpose register conventions.

Table 24. Special-purpose register usage across calls

<table>
<thead>
<tr>
<th>Register</th>
<th>Preserved Across Calls</th>
</tr>
</thead>
</table>
| Condition register  
  Bits 0-7  (CR0,CR1)  
  Bits 8-22 (CR2,CR3,CR4)  
  Bits 23-31 (CR5,CR6,CR7)  | no  
  yes  
  no  
| Link register  | no  |
| 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.
## Runtime Stack for 32-bit Environment

**Vector Information not Included**

<table>
<thead>
<tr>
<th>Low Addresses</th>
<th>Stack grows at this end.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Callee's stack --&gt; 0</td>
<td>Back chain</td>
</tr>
<tr>
<td>8</td>
<td>Saved CR</td>
</tr>
<tr>
<td>12-16</td>
<td>Reserved</td>
</tr>
<tr>
<td>20</td>
<td>Saved TOC</td>
</tr>
<tr>
<td>Space for P1-P8 is always reserved</td>
<td>OUTPUT ARGUMENT AREA</td>
</tr>
<tr>
<td></td>
<td>&lt;--(Used by callee to construct argument list)</td>
</tr>
<tr>
<td></td>
<td>Calllee's stack area</td>
</tr>
<tr>
<td></td>
<td>&lt;-- LOCAL STACK AREA</td>
</tr>
<tr>
<td></td>
<td>(Possible word wasted for alignment.)</td>
</tr>
<tr>
<td>-8<em>nfprs-4</em>ngprs --&gt; save</td>
<td>Save area for caller's GPR</td>
</tr>
<tr>
<td></td>
<td>max 19 words</td>
</tr>
<tr>
<td>-8*nfprs --&gt;</td>
<td>Save area for caller's FPR</td>
</tr>
<tr>
<td></td>
<td>max 18 dblwds</td>
</tr>
<tr>
<td>Caller's stack --&gt; 0</td>
<td>Back chain</td>
</tr>
<tr>
<td>4</td>
<td>Saved CR</td>
</tr>
<tr>
<td>8</td>
<td>Saved LR</td>
</tr>
<tr>
<td>12-16</td>
<td>Reserved</td>
</tr>
<tr>
<td>20</td>
<td>Saved TOC</td>
</tr>
<tr>
<td>Space for P1-P8 is always reserved</td>
<td>INPUT PARAMETER AREA</td>
</tr>
<tr>
<td>24</td>
<td>P1</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>Pn</td>
</tr>
<tr>
<td></td>
<td>Calllee's stack area</td>
</tr>
<tr>
<td>High Addresses</td>
<td></td>
</tr>
</tbody>
</table>
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:

- 18×8 for FPRs + 19×4 for GPRs
- 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.

- 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 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 233 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 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 233 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 (P₁...Pₙ) must be large enough to hold the largest parameter list.

Here is an example of a call to a function:

\[
f(%val(11), %val(12), %val(13), %val(d1), %val(f1),
   %val(c1), %val(d2), %val(s1), %val(cx2))
\]

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 10. Storage mapping of parm area on the stack in 32-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
- “Present” indicators for optional arguments
- Length arguments for strings

Linkage convention for function calls
Function calls to a routine make use of its function descriptor and entry point symbols.

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).

---

1. 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.
2. If foo is bound in the same object module as its caller and a \texttt{LOAD} instruction \texttt{L R2,20(R1)} for 32-bit and \texttt{L R2,40(R1)} for 64-bit, or \texttt{ORI R0,R0,0} immediately follows the \texttt{BL} instruction, the linker replaces the \texttt{LOAD} instruction with a \texttt{NOP (ORI R0,R0,0)}.

\textbf{Note:} For any export, the linker inserts the procedure's descriptor into the object module.

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

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

\begin{verbatim}
bl foo ; branch to function foo
\end{verbatim}

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

\begin{verbatim}
bl 0x60 ; symbol stub for foo
\end{verbatim}

The function stub is located in the \texttt{pic\symbol\_stub} section of the object. The compiler generates a stub for each unique, imported function. The stub contains the instructions that load the address of the routine and branch to it (the routine).

The linker will bypass the stub for functions whose code is contained in the module being produced. It does this by rewriting the \texttt{BL} instruction such that the address of the stub is replaced by the address of the function being called. This means that only calls to dynamic library routines will go through a stub. All other calls will be direct calls in the final executable.

\textbf{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 \texttt{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.

In 32-bit mode, function pointers are 4 bytes long and contain a 32-bit address. In 64-bit mode, they are 8 bytes long and contain a 64-bit address. For pointers to local functions, the address contained is the address of the function in the text section. For imported functions, the address is that of the function's stub. Every unique, imported function will have a stub in the object. The function stub is in the non-lazy symbol pointer section.
Function values

Functions return their values according to type:

- **INTEGER** and **LOGICAL** of kind 1, 2, and 4 are returned (right justified) in R3.
- 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.
- Vector results are returned in VPR2
- 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

You need to consider a number of steps when entering a procedure and when exiting a procedure.

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 8-2319 (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 VPRs that are used by this procedure in the callers VPR save area.
5. Save the VRSAVE register
6. Save all nonvolatile GPRs that are used by this procedure in the caller’s GPR save area.
7. 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 all VPRs saved
3. Restore the VRSAVE register
4. Restore stack pointer to the value it had on entry.
5. Restore link register if necessary.
6. Restore bits 8-2319 of the CR if necessary.
7. If you saved any FPRs, restore them using _restf, where n is the first FPR to be restored.
8. 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.

**Note:** You can make the traceback table smaller or remove it entirely with the **-qtbtable** option

### Stack Size limit

Although the stack’s storage limit is 64 MB, its size defaults to 512 KB. For applications that use large amounts of stack storage, you can set the limit to the maximum allowed.

For example, you can use this **bash** command:

```
ulimit -s 65535
```

Some applications may exceed the allowed maximum. You can try compiling these applications with the **-qsav** compiler option, which defaults to the **STATIC** storage class. However, you should be careful that your program remains semantically correct if **-qsav** is used.

You can also try the **-qsmallstack -qsmallstack=dylenonheap** and **-qstacktemp** options to limit stack usage.

### THREADLOCAL common blocks and interlanguage calls 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, refer to AIX information on threads programming.
Internally, the storage for the thread-specific common block is allocated dynamically by the Fortran runtime library. The Fortran runtime 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:

```fortran
  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 follows in C:

```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.

```fortran
SUBROUTINE fort_sub()
  COMMON /block/ j
  INTEGER :: j
  !IBM* THREADLOCAL /block/  ! Each thread executing fort_sub
  INTEGER a(10)  ! obtains its own copy of /block/.
  
  !IBM* INDEPENDENT
  DO index = 1,10
     CALL another_sub(a(index))
  END DO

END SUBROUTINE fort_sub
```

```fortran
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 "THREADLOCAL" on page 106.
Chapter 9. Implementation details of XL Fortran Input/Output (I/O)

This topic describes XL Fortran support (through extensions and platform-specific details) for the AIX file system.

See “Mixed-language input and output” on page 214 for further considerations related to input and output operations.

Implementation details of file formats

The manner in which XL Fortran implements files is based on their file format.

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. For 64-bit applications, the length of the integer is 4 bytes if you set the uwidth runtime option to 32 (the default), and 8 bytes if you set the uwidth runtime option to 64.

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 operating system 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 XL Fortran Internal limits in the XL Fortran Compiler Reference.)
• 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

There are a number of considerations to be aware of when working with file names.

You can specify file names as either relative (such as file, dir/file, or ..file) or absolute (such as /file or /dir/file). The maximum length of a file name (the full path name) is 1023 characters, even if you only specify a relative path name in the I/O statement. The maximum length of a file name with no path is 255 characters. File names on AIX are case-insensitive. The files test.f and test.F are considered to have the same name and cannot be in the same directory. The XL Fortran compiler, however, recognizes case-sensitive file names.

You must specify a valid file name in such places as the following:
• The FILE= specifier of the OPEN and INQUIRE statements
• INCLUDE lines

Note: To specify a file whose location depends on an environment variable, you can use the GET_ENVIRONMENT_VARIABLE intrinsic procedure to retrieve the value of the environment variable:

```fortran
character(100) home, name
 call get_environment_variable('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

Whether files are preconnected or implicitly connected files is dependent on their units and specific statements.

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=, DECIMAL= or PAD= specifiers.

You can also implicitly connect units 5 and 6 (and *) by using I/O statements that follow a CLOSE:

```fortran
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:

```bash
ln myfile fort.10
```

When you run an application that uses the implicitly connected 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:

```bash
rm fort.10
```

File positioning

The position of a file pointer when a file is opened with no POSITION= specifier is summarized in the following table.
The important things to note are:

- **1** The behavior of newer commands like `xlf95` or `xlf2003` 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 a newer command like `xlf95` or `xlf2003`. 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 newer commands like `xlf95` and `xlf2003`.

- **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 newer commands like `xlf95` and `xlf2003`:
  - 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:
You can refer to the following sections of the AIX Commands Reference, Volumes 1 - 6 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 XL Fortran 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**

The default record lengths for files is dependent on the file format and on the RECL= qualifier.

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] runtime option in the XL Fortran Compiler Reference.)

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**

There are various ways to control a program’s behavior when errors are encountered.

By default, an XL Fortran-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 XLRTEOPTS environment variable, which is described in [Setting runtime options] in the XL Fortran Compiler Reference, 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 XLRTEOPTS 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 behavior 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:

```plaintext
# Switch defaults for some runtime settings.
XLRTEOPTS="err_recovery=no:cnverr=no"
export XLRTEOPTS
```

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:

```fortran
PROGRAM RTOPTS
USE XLPUTIL
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 `XLRTEOPTS` environment variable, be sure to use `SETRTEOPTS` to set all the runtime 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 and is used in the following example:

```fortran
INTEGER, PARAMETER :: UNIT = 10
DO I = 1, 1000000
  WRITE(UNIT, *) 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(UNIT, *) I
  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**

"Mixed-language input and output” on page 214

To improve performance, the XL Fortran runtime 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.

See FLUSH in the Compiler 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 runtime 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:

```fortran
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)
```

```fortran
XLFRTDEOPTS="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/Users/username/data" # Somewhere besides /tmp.
export XLFRTDEOPTS XLFUNIT_1 XLFUNIT_12 XLFUNIT_123
```

Note:
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 `XLFRTDEOPTS` variable, depending on what other runtime options you have set. See Setting runtime options in the XL Fortran Compiler Reference for other options that might be part of the `XLFRTDEOPTS` value.
3. If the `unit_vars` runtime 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 runtime 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:

```fortran
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)
```
Note:
1. The XLFSRATCH_number variable name must be in uppercase, and number must not have any leading zeros.
2. scratch_vars=yes might be only part of the value for the XLFRTEOPTS variable, depending on what other runtime options you have set. See Setting runtime options in the XL Fortran Compiler Reference for other options that might be part of the XLFRTEOPTS value.
3. If the scratch_vars runtime option is set to no or is undefined or if the applicable XLFSRATCH_number variable is not set when the program is run, the program chooses a unique file name for the scratch file and puts it in the directory named by the TMPDIR variable or in the /tmp directory if the TMPDIR variable is not set.

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.

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 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/vlv99) 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.

Note:
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 \texttt{-qalign=4k}.

4. Regardless of any \texttt{STATUS='SCRATCH'} or \texttt{STATUS='DELETE'} specifiers,
neither the data in a logical volume nor the special file in /dev is destroyed by
an \texttt{OPEN} or \texttt{CLOSE} statement.

Related reference

See the -qalign option in the Compiler Reference

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 \texttt{Logical volume I/O} on page 245 and set up the logical volume
especially for high-performance striped I/O through the \texttt{smit} or \texttt{mklv} commands.
You can then use the technique that is described in \texttt{Naming scratch files} on page
244 to place a scratch file on a striped logical volume.

For example, consider a Fortran program that contains the following statements:

\begin{verbatim}
OPEN (UNIT=42, STATUS='SCRATCH',
+ FORM='UNFORMATTED', ACCESS='DIRECT', RECL=131072)
...
OPEN (UNIT=101, STATUS='SCRATCH',
+ FORM='UNFORMATTED', ACCESS='DIRECT', RECL=131072)
\end{verbatim}

You could place the scratch files for units 42 and 101 on the raw logical volumes
\texttt{/dev/rlv30} and \texttt{/dev/rlv31} by setting environment variables before running the
program, as follows:

\begin{verbatim}
XLFRTEOPTS="scratch_vars=yes"
XLFSRATCH_42="/dev/rlv30"
XLFSRATCH_101="/dev/rlv31"
export XLFRTEOPTS XLFSRATCH_42 XLFSRATCH_101
\end{verbatim}

\texttt{AIX Performance Management} 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 following topics in the \texttt{XL Fortran Language Reference}:

- \texttt{INQUIRE} Statement
- \texttt{OPEN} Statement
- \texttt{READ} Statement
- \texttt{WAIT} Statement
- \texttt{WRITE} Statement
Execution of an asynchronous data transfer operation

The effect of executing an asynchronous data transfer operation will be as if the following steps were performed in the order specified, with steps (6)-(9) possibly occurring asynchronously:

1. Determine the direction of the data transfer.
2. Identify the unit.
3. Establish the format if one is present.
4. Determine whether an error condition, end-of-file condition, or end-of-record condition has occurred.
5. Cause the variable that you specified in the IOSTAT= specifier in the data transfer statement to become defined.
6. Position the file before you transfer data.
7. Transfer data between the file and the entities that you specified by the input/output list (if any).
8. Determine whether an error condition, end-of-file condition, or end-of-record condition has occurred.
9. Position the file after you transfer data.
10. Cause any variables that you specified in the IOSTAT= and SIZE= specifiers in the WAIT statement to become defined.

Usage

You can use Fortran asynchronous READ and WRITE statements to initiate asynchronous data transfers in Fortran. Execution continues after the asynchronous I/O statement, regardless of whether the actual data transfer has completed.

A program may synchronize itself with a previously initiated asynchronous I/O statement by using a WAIT statement. There are two forms of the WAIT statement:

1. In a WAIT statement without the DONE= specifier, the WAIT statement halts execution until the corresponding asynchronous I/O statement has completed:

   integer idvar
   integer, dimension(1000):: a
   ....
   READ(unit_number,ID=idvar) a
   ....
   WAIT(ID=idvar)
   ....

2. In a WAIT statement with the DONE= specifier, the WAIT statement returns the completion status of an asynchronous I/O statement:

   integer idvar
   logical done
   integer, dimension(1000):: a
   ....
   READ(unit_number,ID=idvar) a
   ....
   WAIT(ID=idvar, DONE=done)
   ....

   The variable you specified in the DONE= specifier is set to "true" if the corresponding asynchronous I/O statement completes. Otherwise, it is set to "false".

The actual data transfer can take place in the following cases:

• During the asynchronous READ or WRITE statement
At any time before the execution of the corresponding WAIT statement
During the corresponding WAIT statement
Because of the nature of asynchronous I/O, the actual completion time of the request cannot be predicted.

You specify Fortran asynchronous READ and WRITE statements by using the ID= specifier. The value set for the ID= specifier by an asynchronous READ or WRITE statement must be the same value specified in the ID= specifier in the corresponding WAIT statement. You must preserve this value until the associated asynchronous I/O statement has completed.

The following program shows a valid asynchronous WRITE statement:

```fortran
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:

```fortran
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:

```fortran
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, al

  ! Opens the file for both synchronous and asynchronous I/O.
  open(20,form="unformatted",access="direct", &
       status="scratch", recl=isize*4,async="yes")

  ! This loop overlaps the initialization of a(:,j) with
  ! asynchronous write statements.
  !
  ! NOTE: The array is written out one column at a time.
  ! Since the arrays in Fortran are arranged in column
  ! major order, each WRITE statement writes out a
  ! contiguous block of the array.
  !
  do 200 j = 1, icol
    a(:,j) = (/ ((i,i=1,isize))/
    write(20, id=handle(j), rec=j) a(:,j)
  200 end do

  ! Wait for all writes to complete before reading.
```

XL Fortran: Optimization and Programming Guide
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 5L™ Version 5.3 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.

By default, AIX supports 16 outstanding asynchronous I/O requests per process, and 90 system wide. When more than 16 outstanding asynchronous I/O requests are submitted, the system returns the Resource temporarily unavailable error (errno 35). Depending on the I/O items specified for an asynchronous I/O statement, there could be multiple asynchronous I/O requests submitted to the system by the XL Fortran runtime implementation. The number of outstanding asynchronous I/O requests allowed by the system can be changed using the `sysctl` utility with the `root` authority. For example:

```
root# sysctl -w kern.aiomax=1000
kern.aiomax: 90 -> 1000
root# sysctl -w kern.aioprocmx=400
kern.aioprocmx: 16 -> 400
root# sysctl -w kern.aiotthreads=8
kern.aiotthreads: 4 -> 8
```

`kern.aiotthreads` specifies the number of asynchronous I/O worker threads. For further details about the `sysctl` utility, refer to AIX information.

**Note:** The setting goes back to the default every time a machine is rebooted. To avoid changing the default manually after every reboot, add the `sysctl` command in the `/etc/rc` file.

**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 -bl://lib/syscalls.exp -lcrypt
```

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 \
   -bl://lib/syscalls.exp -bl://lib/aio.exp -lcrypt
```

Note that the additional options are `-lc` and `-bl://lib/aio.exp`.

The following table summarizes the options that you need to bind applications in different situations:
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** runtime option to **no**. If you set the **err_recovery** runtime 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 runtime library **libxlf90_r.a** provides support for parallel execution of Fortran I/O statements.

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 runtime 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 runtime 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 runtime 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 runtime 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 runtime 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](#) for details.

Parallel I/O issues

The order in which parallel threads perform I/O operations is not predictable. The XL Fortran runtime 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:
  ```fortran
  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 and 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 runtime 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 XL Fortran runtime synchronizes retrieving and updating the status of the logical units while performing data transfer in parallel. In order to increase performance, it is recommended to increase the size of data transfer per I/O request. The do loop, therefore, should be rewritten as follows:

  read(file_unit, *) a(:, thread_id * 2 + 1 : thread_id * 2 + 2)
  do i = 1, max
    ! Do something for each element of array 'aa'.
  end do

- The result does not depend on the order in which records are written out or read in:

  real a(100)
  do i = 1, 10
    read(4) a(i)
  enddo
call qsort_(a)

- Each thread performs I/O on a different logical unit of direct access, sequential access, or stream access:
do i = 1, 500  
  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 runtime 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 runtime is asynchronous signal unsafe. This means that an XL Fortran I/O statement 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 runtime 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 runtime.
Chapter 10. Implementation details of XL Fortran floating-point processing

This topic answers some common questions about floating-point processing.
- 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?

The topics describing floating-point precision make frequent reference to the compiler options that are grouped together in `Floating-point and integer control` in the XL Fortran Compiler Reference, especially the `-qfloat` option. The XL Fortran compiler also provides three intrinsic modules for exception handling and IEEE arithmetic support to help you write IEEE module-compliant code that can be more portable. See `IEEE Modules and Support` in the XL Fortran Language Reference for details.

The use of the compiler options for floating-point calculations affects the accuracy, performance, and possibly the correctness of floating-point calculations. Although the default values for the options were chosen to provide efficient and correct execution of most programs, you may need to specify nondefault options for your applications to work the way you want. We strongly advise you to read this section before using these options.

Notes:
- 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 273 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

The ANSI/IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-1985 and the details of how it applies to XL Fortran on specific hardware platforms, are summarized in the following topics.

For information on the Fortran 2003 IEEE Module and arithmetic support, see the XL Fortran Language Reference.

Compiling for strict IEEE conformance

By default, XL Fortran follows most, but not all of the rules in the IEEE standard. To compile for strict compliance with the standard:
- Use the compiler option `-qfloat=nomaf`. 
• If the program changes the rounding mode at run time, include \texttt{rrm} among the \texttt{-qfloat} suboptions.

• If the data or program code contains signaling NaN values (NaNS), include \texttt{nans} among the \texttt{-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 \texttt{-qinitauto} compiler option.)

• If you are compiling with \texttt{-O3} or a higher base optimization level, include the \texttt{-qstrict} option. You can also use the \texttt{-qstrict} suboptions to refine the level of control for the transformations performed by the optimizers.

Related reference

See \texttt{-qstrict} in the Compiler Reference

\section*{IEEE Single- and double-precision values}

XL Fortran encodes single-precision and double-precision values in IEEE format. For the range and representation, see \texttt{Real} in the \textit{XL Fortran Language Reference}.

\section*{IEEE Extended-precision values}

The IEEE standard suggests, but does not mandate, a format for \texttt{extended-precision} values. XL Fortran does not use this format. \textquote{Extended-precision values} on page 261 describes the format that XL Fortran uses.

\section*{Infinities and NaNs}

For single-precision real values:

• Positive infinity is represented by the bit pattern \texttt{X'7F80 0000'}.  

• Negative infinity is represented by the bit pattern \texttt{X'FF80 0000'}.  

• A signaling NaN is represented by any bit pattern between \texttt{X'7F80 0001'} and \texttt{X'7FBF FFFF'} or between \texttt{X'FF80 0001'} and \texttt{X'FFBF FFFF'}.  

• A quiet NaN is represented by any bit pattern between \texttt{X'7FC0 0000'} and \texttt{X'7FF FFFF'} or between \texttt{X'FFC0 0000'} and \texttt{X'FFFF FFFF'}.  

For double-precision real values:

• Positive infinity is represented by the bit pattern \texttt{X'7FF00000 00000000'}.  

• Negative infinity is represented by the bit pattern \texttt{X'FF00000 00000000'}.  

• A signaling NaN is represented by any bit pattern between \texttt{X'7FF00000 00000001'} and \texttt{X'7FF00 00000000'} or between \texttt{X'7FF00000 00000001'} and \texttt{X'7FF00000 00000001'}.  

• A quiet NaN is represented by any bit pattern between \texttt{X'7FF80000 00000000'} and \texttt{X'7FFFFFF FFFFFFFFFFF'} or between \texttt{X'7FF80000 00000000'} and \texttt{X'7FF00000 00000000'}.  

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 \texttt{ieee\_value} function provided in the \texttt{ieee\_arithmetic} intrinsic module. Using the \texttt{ieee\_value} function is the preferred programming technique, as it is allowed by the Fortran 2003 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.
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 by default does not 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 28. Results of IEEE exceptions, with and without trapping enabled

<table>
<thead>
<tr>
<th>Exceptions not enabled (default)</th>
<th>Overflow</th>
<th>Underflow</th>
<th>Zerodivide</th>
<th>Invalid</th>
<th>Inexact</th>
</tr>
</thead>
<tbody>
<tr>
<td>INF</td>
<td>INF</td>
<td>Denormalized number</td>
<td>INF</td>
<td>NaN</td>
<td>Rounded result</td>
</tr>
<tr>
<td>Exceptions enabled</td>
<td>Unnormalized number with biased exponent</td>
<td>Unnormalized number with biased exponent</td>
<td>No result</td>
<td>No result</td>
<td>Rounded result</td>
</tr>
</tbody>
</table>

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 267 for instructions on doing so.

Hardware-specific floating-point overview

Single- and double-precision values and extended-precision values for hardware-specific floating-point processing are described in the following topics.

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 at which either an individual multiply or add 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 262 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.

**Note:**

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 267) 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 265.

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(fpnr1) and
fpstat(fpnr2) are set as specified in the following table:

<table>
<thead>
<tr>
<th>fpstat(fpnr1)</th>
<th>fpstat(fpnr2)</th>
<th>Rounding Mode Enabled</th>
</tr>
</thead>
<tbody>
<tr>
<td>.true.</td>
<td>.true.</td>
<td>Round towards -infinity.</td>
</tr>
<tr>
<td>.true.</td>
<td>.false.</td>
<td>Round towards +infinity.</td>
</tr>
<tr>
<td>.false.</td>
<td>.true.</td>
<td>Round towards zero.</td>
</tr>
<tr>
<td>.false.</td>
<td>.false.</td>
<td>Round to nearest.</td>
</tr>
</tbody>
</table>

For example:

```fortran
program fptest
  include 'fpdc.h'

  call fpgets( fpstat ) ! Get current register values.
  if ( (fpstat(fpnr1).eqv..false.) .and. +
       (fpstat(fpnr2).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(fpnr1) = .true. ! These 2 lines mean round towards
  fpstat(fpnr2) = .true. ! +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(fpnr1) = .true. ! These 2 lines mean round towards
  fpstat(fpnr2) = .true. ! -INFINITY.
  call fpsets( fpstat )
  r = -2.0 / 3.0
  print *, 'Round towards -INFINITY: -2.0 / 3.0= ', r
end
```

XL Fortran: Optimization and Programming Guide
This block data program unit initializes the fpstat array, and so on.

```fortran
! 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 **fpset** and **fpget** 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:

```fortran
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:

```fortran
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
```

**Note:**
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, xlf95_r7, and xlf2003_r 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 threadsafe, do not specify IBMT as a trigger constant.

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 runtime calculations. However, the results of the compiler’s evaluation might not match exactly the results of the runtime 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 262] with this command if the program consistently uses round-to-plus-infinity mode:

\texttt{xlf95 \textasciitilde qieee\textasciitilde plus \textasciitilde qfloat\textasciitilde rrm changes\_rounding\_mode.f}

### Duplicating the floating-point results of other systems

The default \texttt{-qarch} options for this version of the compiler are different from previous versions. This can affect results when using 4 byte floating point operations and you want the same results as previous versions. For example, POWER and POWER2 architectures only had 8 byte floats and simulation was used to approximate 4 byte floating point operations. To duplicate results for older \texttt{-qarch} settings while using new \texttt{-qarch} settings like \texttt{ppc} or \texttt{pwr6}, you should additionally use \texttt{-qfloat\textasciitilde nosingle} and \texttt{-qfloat\textasciitilde norndsngl}.

To duplicate the double-precision results of programs on systems with different floating-point architectures (without multiply-add instructions), specify the \texttt{nomaf} suboption of the \texttt{-qfloat} option. This suboption prevents the compiler from generating any multiply-add instructions. 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 \texttt{REAL} items is different from that on systems running XL Fortran, use the \texttt{-qrealsize} option to change the default \texttt{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 \texttt{DOUBLE PRECISION} variables, use the \texttt{-qdpc} or \texttt{-qrealsize} option.

If results consistent with other systems are important to you, include \texttt{norsqrt} and \texttt{nofold} in the settings for the \texttt{-qfloat} option. If you specify the option \texttt{-O3}, include \texttt{-qstrict} and any necessary suboptions too.

**Related information**

- See \texttt{-qarch} in the Compiler Reference
- See \texttt{-qfloat} in the Compiler Reference
- See \texttt{-qrealsize} in the Compiler Reference
- See \texttt{-qstrict} in the Compiler Reference

### 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 \texttt{-O} option, and specify \texttt{-qfloat\textasciitilde rsqrt\textasciitilde hssngl\textasciitilde f1tint}.

The following section describes the functions of these suboptions:
• The **sqrt** 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 **hsngl** suboption is the opposite of **rndsn**; 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, **hsngl** 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 **hsngl** 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 **flint** 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 **hsngl** and **flint** with the **hsflt** suboption; it does the same thing as **flint** and suppresses rounding operations.

In suppressing rounding operations, **hsflt** works like **hsngl**, 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 attempt to convert floating-point values to integer when the floating-point values are outside the range of the corresponding integer types.

• 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 **hsngl** instead.

[Technical details of the -qfloat=hsflt option](https://www.ibm.com/support/knowledgecenter/SSEQVZ_11.1.0/xcsp/langinfo/techdetail.html) in the XL Fortran Compiler Reference 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 topics 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, provided that a signal handler for SIGTRAP is installed.

-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.
Installing exceptions

Operating system features for trapping floating-point exceptions

A direct way to turn on exception trapping is to call the operating system routine \texttt{fp\_trap}. It uses the system hardware to detect floating-point exceptions and generates \texttt{SIGPE} signals when exceptions occur. Fortran definitions for the values needed to call it are in the files \texttt{/usr/include/fp\_fort\_c.f}, \texttt{fp\_fort\_t.f}, or the \texttt{xlf\_fp\_util} module.

There are other related operating system routines that you can locate by reading the description of \texttt{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 \texttt{SIGPE} signals, the same as other popular UNIX\textsuperscript{®} systems.

The disadvantages of this approach include:

- The program may run much slower while exception checking is turned on.
- The call to \texttt{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 \texttt{-qsigtrap} option, applies both to \texttt{SIGTRAP} and \texttt{SIGPE} 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 \texttt{SIGTRAP} signal handler, use the \texttt{-qsigtrap} option. It allows you to specify an XL Fortran handler that produces a traceback or to specify a handler you have written:

\begin{verbatim}
xlf95 -qflttrap=ov:und:en pl.f     # Dump core on an exception
xlf95 -qflttrap=ov:und:en -qsigtrap pi.f    # Uses the xl_trc handler
xlf95 -qflttrap=ov:und:en -qsigtrap=return_22_over_7 pi.f # Uses any other handler
\end{verbatim}

You can also install an alternative exception handler, either one supplied by XL Fortran or one you have written yourself, by calling the \texttt{SIGNAL} subroutine (defined in \texttt{/usr/include/fexcept.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**
Producer 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_sigmune**
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/signal.h` defines a Fortran derived type similar to the sigcontext structure in . You can write a Fortran signal handler that accesses this derived type.

“Sample programs for exception handling” on page 272 lists some sample programs that illustrate how to use these signal handlers or write your own. Also see the [SIGNAL procedure](#) in the XL Fortran Language Reference for more information.

**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 `-qfptrap` 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 `-qfptrap` option:
Check all exceptions, 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 262.

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:

```fortran
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:

```fortran
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 at link time. The following files are supplied for the xlf_fp_util module:

<table>
<thead>
<tr>
<th>File names</th>
<th>File type</th>
<th>Locations</th>
</tr>
</thead>
<tbody>
<tr>
<td>xlf_fp_util.mod</td>
<td>module symbol file</td>
<td>/usr/lpp/xlf/include_d7</td>
</tr>
<tr>
<td></td>
<td>(32-bit)</td>
<td>/usr/lpp/xlf/include</td>
</tr>
<tr>
<td></td>
<td>module symbol file</td>
<td>/usr/lpp/xlf/include</td>
</tr>
<tr>
<td></td>
<td>(64-bit)</td>
<td></td>
</tr>
</tbody>
</table>
To use the procedures, you must add a **USE XLF_FP_UTIL** statement to your source file. For more information, see the **USE statement** in the *XL Fortran Language Reference*.

When compiling with the -U option, you must code the names of these procedures in all lowercase.

For a list of the *xlf_fp_util* procedures, see the **Service and utility procedures** section in the *XL Fortran Language Reference*.

### fpgets and fpsets subroutines

The *fpsets* and *fpgets* subroutines provide a way to manipulate or query the floating-point status and control register. Instead of calling the operating system routines directly, you pass information back and forth in *fpstat*, an array of logicals. The following table shows the most commonly used array elements that deal with exceptions:

**Table 30. Exception bits to use with fpsets and fpgets**

<table>
<thead>
<tr>
<th>Array Element to Set to Enable</th>
<th>Array Element to Check if Exception Occurred</th>
<th>Exception Indicated When .TRUE.</th>
</tr>
</thead>
<tbody>
<tr>
<td>n/a</td>
<td>fpstat(fpfx)</td>
<td>Floating-point exception summary</td>
</tr>
<tr>
<td>n/a</td>
<td>fpstat(fpfex)</td>
<td>Floating-point enabled exception summary</td>
</tr>
<tr>
<td>fpstat(fpve)</td>
<td>fpstat(fpvx)</td>
<td>Floating-point invalid operation exception summary</td>
</tr>
<tr>
<td>fpstat(fpoe)</td>
<td>fpstat(fpox)</td>
<td>Floating-point overflow exception</td>
</tr>
<tr>
<td>fpstat(fpue)</td>
<td>fpstat(fpux)</td>
<td>Floating-point underflow exception</td>
</tr>
<tr>
<td>fpstat(fpze)</td>
<td>fpstat(fpzx)</td>
<td>Zero-divide exception</td>
</tr>
<tr>
<td>fpstat(fpxe)</td>
<td>fpstat(fpxx)</td>
<td>Inexact exception</td>
</tr>
<tr>
<td>fpstat(fpve)</td>
<td>fpstat(fpvxsnan)</td>
<td>Floating-point invalid operation exception (NaNS)</td>
</tr>
<tr>
<td>fpstat(fpve)</td>
<td>fpstat(fpvxisi)</td>
<td>Floating-point invalid operation exception (INF-INF)</td>
</tr>
<tr>
<td>fpstat(fpve)</td>
<td>fpstat(fpvxdid)</td>
<td>Floating-point invalid operation exception (INF/INF)</td>
</tr>
<tr>
<td>fpstat(fpve)</td>
<td>fpstat(fpvxdz)</td>
<td>Floating-point invalid operation exception (0/0)</td>
</tr>
<tr>
<td>fpstat(fpve)</td>
<td>fpstat(fpvximz)</td>
<td>Floating-point invalid operation exception (INF*0)</td>
</tr>
<tr>
<td>fpstat(fpve)</td>
<td>fpstat(fpvxvc)</td>
<td>Floating-point invalid operation exception (invalid compare)</td>
</tr>
<tr>
<td>n/a</td>
<td>fpstat(fpvxsoft)</td>
<td>Floating-point invalid operation exception (software request), PowerPC only</td>
</tr>
<tr>
<td>n/a</td>
<td>fpstat(fpvxsqrt)</td>
<td>Floating-point invalid operation exception (invalid square root), PowerPC only</td>
</tr>
<tr>
<td>n/a</td>
<td>fpstat(fpvxcvi)</td>
<td>Floating-point invalid operation exception (invalid integer convert), PowerPC only</td>
</tr>
</tbody>
</table>
To explicitly check for specific exceptions at particular points in a program, use \texttt{fpgets} and then test whether the elements in \texttt{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 \texttt{fpstat(fpx)}, \texttt{fpstat(fvx)}, and \texttt{fpstat(fpfx)}, which are reset only when the specific exception bits are reset.

An advantage of using the \texttt{fpgets} and \texttt{fpsets} subroutines (as opposed to controlling everything with suboptions of the \texttt{-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:

\begin{itemize}
  \item You have to change your source code.
  \item These routines differ from what you may be accustomed to on other platforms.
\end{itemize}

For example, to trap floating-point overflow exceptions but only in a certain section of the program, you would set \texttt{fpstat(fpoe)} to \texttt{.TRUE.} and call \texttt{fpsets}. After the exception occurs, the corresponding exception bit, \texttt{fpstat(fpox)}, is \texttt{.TRUE.} until the program runs:

\begin{verbatim}
  call fpgets(fpstat)
  fpstat(fpox) = .FALSE.
  call fpsets(fpstat)  ! resetting fpstat(fpox) to .FALSE.
\end{verbatim}

\section*{Sample programs for exception handling}
\texttt{/usr/lpp/xlf/samples/floating_point} contains a number of sample programs to illustrate different aspects of exception handling:

\texttt{flttrap_handler.c} and \texttt{flttrap_test.f}
A sample exception handler that is written in C and a Fortran program that uses it.

\texttt{xl_ieee.F} and \texttt{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.

\texttt{check_fpsets.f} and \texttt{postmortem.f}
Show how to work with the \texttt{fpsets} and \texttt{fpgets} procedures and the \texttt{fpstats} array.

\texttt{fhandler.F}
Shows a sample Fortran signal handler and demonstrates the \texttt{xl_sigdump} procedure.

\texttt{xl_trbk_test.f}
Shows how to use the \texttt{xl_trbk} procedure to generate a traceback listing without stopping the program.

The sample programs are strictly for illustrative purposes only.

\section*{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.

**Note:** 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.

**Note:**
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

This version of XL Fortran no longer supports the **-qarch** suboptions to target these processors. If you are concerned about reproducing results similar to these architectures on the currently supported PowerPC targets, consider using the **-qfloat=nosingle** and **-qfloat=normdsngl** options together.
**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:

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 `norndsngl` 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:

```fortran
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 the COM architecture. `rndsngl` is the default when you target a PowerPC architecture. You can also explicitly set the `rdmsngl` 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 number.

---

XL Fortran: Optimization and Programming Guide
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.
Chapter 11. 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 of porting a typical program is described in this topic.

The process for porting a typical program looks like this:

1. Identify any nonportable language extensions or subroutines that you used in the original program. Check to see which of these XL Fortran supports:
   - Language extensions are identified in the XL Fortran Language Reference.
   - Some extensions require you to specify an XL Fortran compiler option; you can find these options listed in the Portability and migration options table in the XL Fortran Compiler Reference.
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 exact bit-pattern 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 XL Fortran-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 12.1.

You can link existing FORTRAN 77 object code from previous version of XL Fortran into programs generated by XL Fortran Version 12.1. See Linking new objects with existing ones in the XL Fortran Compiler Reference 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 \textit{trigger\_constants} other than the defaults in XL Fortran, you can use the \texttt{-qdirective} compiler option to specify them. For instance, if you are porting CRAY code contained in a file \texttt{xx.f}, you would use the following command to add the CRAY \textit{trigger\_constant}:

\begin{verbatim}
xlf95 xx.f -qdirective=mic$
\end{verbatim}

For fixed source form code, in addition to the \texttt{!} value for the \textit{trigger\_head} portion of the directive, XL Fortran also supports the \textit{trigger\_head} values \texttt{C}, \texttt{c}, and \texttt{*}.

For more information, see the \texttt{-qdirective} option in the \textit{XL Fortran Compiler Reference}.

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 31. \texttt{PARALLEL DO} Clauses and their XL Fortran synonyms

<table>
<thead>
<tr>
<th>\texttt{PARALLEL DO} Clause</th>
<th>XL Fortran Synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>LASTLOCAL</td>
<td>LASTPRIVATE</td>
</tr>
<tr>
<td>LOCAL</td>
<td>PRIVATE</td>
</tr>
<tr>
<td>\texttt{MP_SCHEDTYPE} \texttt{and CHUNK}</td>
<td>SCHEDULE</td>
</tr>
<tr>
<td>SAVELAST</td>
<td>LASTPRIVATE</td>
</tr>
<tr>
<td>SHARE</td>
<td>SHARED</td>
</tr>
<tr>
<td>NEW</td>
<td>PRIVATE</td>
</tr>
</tbody>
</table>

### Table 32. \texttt{PARALLEL DO} scheduling types and their XL Fortran synonyms

<table>
<thead>
<tr>
<th>Scheduling Type</th>
<th>XL Fortran Synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>GSS</td>
<td>GUIDED</td>
</tr>
<tr>
<td>\texttt{INTERLEAVE}</td>
<td>\texttt{STATIC}(1)</td>
</tr>
<tr>
<td>\texttt{INTERLEAVED}</td>
<td>\texttt{STATIC}(1)</td>
</tr>
<tr>
<td>\texttt{INTERLEAVE}(n)</td>
<td>\texttt{STATIC}(n)</td>
</tr>
<tr>
<td>\texttt{INTERLEAVED}(n)</td>
<td>\texttt{STATIC}(n)</td>
</tr>
<tr>
<td>SIMPLE</td>
<td>STATIC</td>
</tr>
</tbody>
</table>

### Table 33. \texttt{PARALLEL SECTIONS} clauses and their XL Fortran synonyms

<table>
<thead>
<tr>
<th>\texttt{PARALLEL SECTIONS} Clause</th>
<th>XL Fortran Synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOCAL</td>
<td>PRIVATE</td>
</tr>
<tr>
<td>SHARE</td>
<td>SHARED</td>
</tr>
<tr>
<td>NEW</td>
<td>PRIVATE</td>
</tr>
</tbody>
</table>

### Common industry extensions that XL Fortran supports

XL Fortran allows many of the same \texttt{FORTRAN 77} extensions as other popular compilers.
These extensions include:

<table>
<thead>
<tr>
<th>Extension</th>
<th>Refer to XL Fortran Language Reference Section(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Typeless constants</td>
<td>Typeless literal constants</td>
</tr>
<tr>
<td>*len length specifiers for types</td>
<td>Data types</td>
</tr>
<tr>
<td>BYTE data type</td>
<td>Byte</td>
</tr>
<tr>
<td>Long variable names</td>
<td>Names</td>
</tr>
<tr>
<td>Lower case</td>
<td>Names</td>
</tr>
<tr>
<td>Mixing integers and logics (with -qintlog option)</td>
<td>Evaluation of expressions</td>
</tr>
<tr>
<td>Character-count Q edit descriptor (with -qqcount option)</td>
<td>Q (Character Count) Editing</td>
</tr>
<tr>
<td>Intrinsic for counting set bits in registers and determining data-object parity</td>
<td>POPCNT</td>
</tr>
<tr>
<td>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)</td>
<td>Integer</td>
</tr>
<tr>
<td>Integer POINTERs, 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.)</td>
<td>POINTER(integer)</td>
</tr>
<tr>
<td>Conditional vector merge (CVMGx) intrinsic functions</td>
<td>CVMGx (TSOURCE, FSOURCE, MASK)</td>
</tr>
<tr>
<td>Date and time service and utility functions (rtc, irtc, jdate, clock, timef, and date)</td>
<td>Service and utility procedures</td>
</tr>
<tr>
<td>STRUCTURE, UNION, and MAP constructs</td>
<td>Structure components (TSOURCE, FSOURCE, MASK)</td>
</tr>
</tbody>
</table>

**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 reference**

See -langlvl in the Compiler Reference
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, Fortran 95 and Fortran 2003 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 265.

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 in the Compiler Reference
- See -qport in the Compiler Reference
Chapter 12. Sample Fortran programs

The programs in the topics referenced here are provided as coding examples for XL Fortran.

Other examples can be found in the /usr/lpp/xlf/samples directory. These illustrate various aspects of XL Fortran programming. 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

This is an example of an 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.0.0) THEN
    Y=X**3
    SUM=SUM+Y
  END IF
END DO
WRITE(*) 'This is the sum of the positive cubes:',SUM
END
```

Execution results

Running the program yields the following results:

```
$ 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 an example of a valid C routine source file used to execute Fortran test subroutines.

```
/*
  * ********************************************
  * This is a main function that creates threads to execute the Fortran
```

© Copyright IBM Corp. 1996, 2008
* 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 <\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
Example 3 - valid Fortran SMP source file

This is an example of a valid Fortran SMP source file used to calculate the value of pi.

```fortran
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

This is an example of an invalid Fortran SMP source file.

```fortran
!* 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  *
```
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)

  integer aa
  !IBM* THREADLOCAL /block/ ! Each thread obtains a new copy
  ! of the common block /block/.
  integer :: j
  aa = j ! The value of "j" is undefined.
end subroutine another_sub

Programming examples using the Pthreads library module

These examples demonstrate the use of the Pthreads library module.

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.

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.
!* 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
Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user’s responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.
Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Lab Director
IBM Canada Ltd. Laboratory
8200 Warden Avenue
Markham, Ontario L6G 1C7
Canada

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM’s future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM’s application programming interfaces.
Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. 1998, 2008. All rights reserved.

This software and documentation are based in part on the Fourth Berkeley Software Distribution under license from the Regents of the University of California. We acknowledge the following institution for its role in this product’s development: the Electrical Engineering and Computer Sciences Department at the Berkeley campus.

### Trademarks and service marks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A complete and current list of IBM trademarks is available on the Web at [http://www.ibm.com/legal/copytrade.shtml](http://www.ibm.com/legal/copytrade.shtml).

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

Cell Broadband Engine is a trademark of Sony Computer Entertainment, Inc. in the United States, other countries, or both and is used under license therefrom.

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

Other company, product, or service names may be trademarks or service marks of others.
Directives (continued)
CRITICAL 71
DO (worksharing) 73
END CRITICAL 71
END DO (worksharing) 73
DO (worksharing) 73
DO SERIAL compiler directive 76
double-precision values 258, 260
dynamic linking
code size 40

E
enable suboption of -qflttrap 269
enabling MASS 9
enabling VMX 9
END CRITICAL 71
END DO (worksharing) 73
END MASTER compiler directive 80
END ORDERED compiler directive 81
END PARALLEL compiler directive 83
END PARALLEL DO compiler directive 86
END PARALLEL SECTIONS compiler directive 90
END PARALLEL WORKSHARE compiler directive 97
END SECTIONS compiler directive 97
environment variables
OpenMP
OMP_DYNAMIC 63
OMP_NUM_THREADS 63
OMP_SCHEDULE 64
runtime
XL/SMP/POTS 58
exception handling 259
for floating point 267
installing an exception handler 268
execution environment routines
OpenMP 136
explicit interfaces 222
extended-precision values 261
extensions to FORTRAN 77, list of common ones 279

F
f_makeitme function 157
f_pthread 154
f_pthread_attr_destroy function 157
f_pthread_attr_getdetachstate function 158
f_pthread_attr_getguardsize function 159
f_pthread_attr_getinitberitesch function 159
f_pthread_attr_getschedparam function 160
f_pthread_attr_getschedpolicy function 160
f_pthread_attr_getscope function 161
f_pthread_attr_getstackaddr function 162
f_pthread_attr_getstacksize function 162
f_pthread_attr_init function 163
f_pthread_attr_setdetachstate function 164
f_pthread_attr_setguardsize 164
f_pthread_attr_setinitberitesch function 165
f_pthread_attr_setschedparam function 166
f_pthread_attr_setschedpolicy function 166
f_pthread_attr_setscope function 167
f_pthread_attr_setstackaddr function 168
f_pthread_attr_setstacksize function 168
f_pthread_attr_t function 169
f_pthread_cancel function 169
f_pthread_cleanup_pop function 170
f_pthread_cleanup_push function 170
f_pthread_cond_broadcast function 172
f_pthread_cond_destroy function 172
f_pthread_cond_init function 173
f_pthread_cond_signal function 173
f_pthread_cond_timedwait function 174
f_pthread_cond_wait function 175
f_pthread_condattr_destroy function 176
f_pthread_condattr_getshared function 176
f_pthread_condattr_init function 177
f_pthread_condattr_setshared function 177
f_pthread_condattr_sett function 178
f_pthread_create function 178
f_pthread_detach function 180
f_pthread_equal function 180
f_pthread_exit function 181
f_pthread_getconcurrency function 181
f_pthread_getschedparam function 182
f_pthread_getspecific function 183
f_pthread_join function 183
f_pthread_key_create function 184
f_pthread_key_delete function 185
f_pthread_key_t 185
f_pthread_kill function 185
f_pthread_mutex_destroy function 186
f_pthread_mutex_getupc eiling function 187
f_pthread_mutex_init function 187
f_pthread_mutex_lock function 188
f_pthread_mutex_setp nceiling function 188
f_pthread_mutex_t 189
f_pthread_mutex_trylock function 189
f_pthread_mutex_unlock function 190
f_pthread_mutexattr_destroy function 191
f_pthread_mutexattr_getpactorol function 191
f_pthread_mutexattr_getpshared function 192
f_pthread_mutexattr_gettype function 193
f_pthread_mutexattr_init function 194
f_pthread_mutexattr_setp nceiling function 194
f_pthread_mutexattr_setprotocol function 195
f_pthread_mutexattr_setupsetshared function 195
f_pthread_mutexattr_settype function 196
f_pthread_mutexattr_settype function 197
f_pthread_once function 197
f_pthread_once_t 198
f_pthread_rwlock_destroy function 198
f_pthread_rwlock_init function 198
f_pthread_rwlock_rdlock function 199
f_pthread_rwlock_rwlock_t function 200
f_pthread_rwlock_tryrdlock function 200
f_pthread_rwlock_trywrlock function 201
f_pthread_rwlock_unlock function 202
f_pthread_rwlock_wllock function 202
f_pthread_rwlockattr_destroy function 203
f_pthread_rwlockattr_getpshared function 203
f_pthread_rwlockattr_init function 204
f_pthread_rwlockattr_setp nceiling function 205
f_pthread_rwlockattr_t function 205
f_pthread_self function 206
f_pthread_setcancelstate function 206
f_pthread_setcanceltype function 207
f_pthread_setconcurrency function 208
f_pthread_setschedparam function 208
f_pthread_setspecific function 209
f_pthread_t function 210
f_pthread_testcancel function 210
f_sched_param function 210
f_sched_yield function 211
f timespec function 211
f77 command
and file positioning 240
fexcept.h include file 269
fhandler.F sample file 272
file positioning 240
files
I/O formats 237
names 238
permissions 242
floating-point exceptions 267
processing 257
optimizing 265
floating-point optimization 26
floating-point status and control
register 271
FLTRAP @PROCESS directive 267
flushinglibc and filterp_test.f sample files 272
FLUSH compiler directive 77
flushing I/O buffers 243
formats, file 237
fort.* default file names 239, 244
FORTRAN 77 extensions, list of common ones 279
f_p_fort.c and f_p_fort.t include files 268
fptrap libc routine 268
fpd.h and fpdch.h include files 262
fpgets and fpsets service and utility subroutines 271
optimizing (continued) 8
option driven tuning 25
OPTIONAL attribute 223
options for targeting your architecture 21
ORDERED compiler directive 81

P
PARALLEL compiler directive 83
PARALLEL DO compiler directive 86
SCHEDULE clause 86
parallel execution options 60
PARALLEL SECTIONS compiler directive 90
PARALLEL WORKSHARE compiler directive 93
parthds runtime option 60
parthreshold runtime option 62
Pascal language and interlanguage calls 213
PDF 13
walkthrough 13
performance analysis 29
performance of floating-point arithmetic 265
performance tuning options 61
permissions of files 242
pipes, interaction of XL Fortran I/O with 241
plus infinity, representation of 258
pointers (integer POINTER) (FORTRAN 77 extension) 279
portability 278
porting to XL Fortran 277
POSITION @PROCESS directive 240
position of a file after an OPEN statement 240
positive infinity, representation of 258
postmortem.f sample file 272
preconnected files 239
profilefreq runtime option 62
providing your application characteristics 26
pseudo-devices, interaction of XL Fortran I/O with 241
pthreads library module 287

Pthreads Library Module descriptions of functions in 154
f_makefile function 157
f_pthread_attr_destroy function 157
f_pthread_attr_getdetachstate function 158
f_pthread_attr_getguardsize function 159
f_pthread_attr_getinheritsched function 159
f_pthread_attr_getschedparam function 159
f_pthread_attr_getschedpolicy function 160
f_pthread_attr_getspecific function 160
f_pthread_attr_getstackaddr function 162
f_pthread_attr_getstacksize function 162
f_pthread_attr_init function 163
f_pthread_attr_setdetachstate function 164
f_pthread_attr_setguardsize function 164
f_pthread_attr_setinheritsched function 165
f_pthread_attr_setschedparam function 166
f_pthread_attr_setschedpolicy function 166
f_pthread_atexit_destroy function 167
f_pthread_atexit_setscheduler function 167
f_pthread_attr_setschedparam function 168
f_pthread_attr_setschedpolicy function 168
f_pthread_attr_setschedparam function 169
f_pthread_atexit_t function 169
f_pthread_cancel function 169
f_pthread_cleanup_pop function 170
f_pthread_cleanup_push function 170
f_pthread_cond_broadcast function 172
f_pthread_cond_destroy function 172
f_pthread_cond_init function 173
f_pthread_cond_signal function 173
f_pthread_cond_timedwait function 174
f_pthread_cond_wait function 174
f_pthread_condattr_destroy function 176
f_pthread_condattr_getshared function 176
f_pthread_condattr_init function 177
f_pthread_condattr_setshared function 177
f_pthread_condattr_settype function 177
f_pthread_cond_t function 178
f_pthread_create function 178
f_pthread_detach function 180
f_pthread_equal function 180
f_pthread_exit function 181
f_pthread_getconcurrency function 181
f_pthread_getschedparam function 182
f_pthread_getspecific function 183
f_pthread_join function 183
f_pthread_key_create function 184
f_pthread_key_delete function 185
f_pthread_key_t function 185
f_pthread_key_destroy function 185
f_pthread_mutex_destroy function 186
f_pthread_mutex_getprioceiling function 187
f_pthread_mutex_init function 187
f_pthread_mutex_lock function 188
f_pthread_mutex_t function 188
f_pthread_mutex_trylock function 189
f_pthread_mutexattr_destroy function 190
f_pthread_mutexattr_destroy function 190
f_pthread_mutexattr_destroy function 191
f_pthread_mutexattr_destroy function 191
f_pthread_mutexattr_destroy function 192
f_pthread_mutexattr_destroy function 193
f_pthread_mutexattr_destroy function 194
f_pthread_mutexattr_destroy function 194
f_pthread_mutexattr_destroy function 195
f_pthread_mutexattr_destroy function 195
f_pthread_mutexattr_destroy function 196
f_pthread_mutexattr_destroy function 196
f_pthread_mutexattr_destroy function 197
f_pthread_mutexattr_destroy function 197
f_pthread_mutexattr_destroy function 198
f_pthread_mutexattr_destroy function 198
f_pthread_mutexattr_destroy function 199
f_pthread_mutexattr_destroy function 199
f_pthread_mutexattr_destroy function 200
f_pthread_mutexattr_destroy function 200
f_pthread_mutexattr_destroy function 200
f_pthread_mutexattr_destroy function 201
f_pthread_mutexattr_destroy function 201
f_pthread_rwlock_destroy function 202
f_pthread_rwlock_destroy function 202
f_pthread_rwlock_destroy function 203
f_pthread_rwlock_destroy function 203
f_pthread_rwlock_destroy function 203
f_pthread_rwlock_destroy function 204
f_pthread_rwlock_destroy function 204
f_pthread_rwlock_destroy function 205
f_pthread_rwlock_destroy function 205
f_pthread_rwlock_destroy function 206
f_pthread_rwlock_destroy function 206
f_pthread_rwlock_destroy function 207
f_pthread_rwlock_destroy function 207
f_pthread_rwlock_destroy function 208
f_pthread_rwlock_destroy function 208
f_pthread_rwlock_destroy function 209
f_pthread_rwlock_destroy function 209
f_pthread_rwlock_destroy function 210
f_pthread_rwlock_destroy function 210
f_pthread_rwlock_destroy function 211
f_pthread_rwlock_destroy function 211

Pthreads Library, AIX 154
Q
Q (character-count) edit descriptor (FORTRAN 77 extension) 279
quiet NaN 258

R
raw logical volumes, I/O operations on 245
real arithmetic 257
REAL(16) values 261
REAL(4) and REAL(8) values 258, 260
record lengths 242
redirecting input/output 240
reference, passing arguments by 221
rounding 262
rounding errors 264
rounding mode 262, 264

S
sample programs 283
calling C functions from Fortran 217
floating-point exception handling 272
for SMP 283
scalar MASS library 45
SCHEDULE clause, of PARALLEL DO directive 86
SCHEDULE compiler directive 94
discussion 94
schedule runtime option 58
scheduling, block cyclic 134
scope
data scope attribute clauses 115
scratch file directory 244
scratch_vars runtime option 244
SECTIONS compiler directive discussion 97
setjthreshold runtime option 62
short vectorization 9
SIGFPE signal 267, 268
signal handling
for floating point 267
installing an exception handler 268
signaling NaN 258, 272
SIGTRAP signal 267, 268
SINGLE / END SINGLE compiler directive 100
single-precision values 258, 260
SMP
sample programs 283
special files, interaction of XL Fortran I/O with 241
spins runtime option 61
stack 225
stack runtime option 60
standard error, input, and output streams 239
star length specifiers 279
static linking
code size 40
status and control register for floating point 271
stdout, stdin, and stdout streams 239
strings, passing to C functions 219
subprograms in other languages, calling 213, 216
Sun pointer (FORTRAN 77 extension), XL Fortran equivalent 279
symbolic links, interaction of XL Fortran I/O with 241
system Pthreads Library 154

T
tape files, interaction of XL Fortran I/O with 241
tcl command 241
thread visible variables 78
thread-safing
-threads library module 154
THREADLOCAL compiler directive 106
THREADPRIVATE compiler directive 108
throughput for I/O, increasing with data striping 245
time and date functions (FORTRAN 77 extension) 279
Trace/BPT trap 268
traceback listing 269
tuning 21
-qarch 22
-qcache 25
-qtune 24
controlling optimization transformations 28
instruction group support by architecture 22
option driven tuning 25
options to avoid 30
performance analysis 29
providing your application characteristics 26
typeless constants (FORTRAN 77 extension) 279

U
unit_vars runtime option 244
usrblds runtime option 60

V
VALUE attribute 221
value, passing arguments by 221
vector MASS library 47
vectorization
long 9
short 9

W
work-sharing constructs
SECTIONS / END SECTIONS compiler directives 97
SINGLE / END SINGLE compiler directives 100
WORKSHARE compiler directive 113

Y
yields runtime option 61