

IBM XL Fortran Advanced Edition for Linux, V11.1



Optimization and Programming Guide

IBM XL Fortran Advanced Edition for Linux, V11.1



Optimization and Programming Guide

Note!

Before using this information and the product it supports, be sure to read the general information under “Notices” on page 255.

First Edition

This edition applies to IBM XL Fortran Advanced Edition for Linux, V11.1 (Program number 5724-S74) 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 1990, 2007. All rights reserved.

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

Contents

About this document vii

Who should read this document	vii
How to use this document	vii
How this document is organized	vii
Conventions and terminology used in this document.	viii
Related information.	xi
IBM XL Fortran publications	xi
Standards and specifications documents.	xii
Other IBM publications	xiii
Technical support	xiii
How to send your comments	xiii

Chapter 1. Optimizing your applications 1

Distinguishing between optimization and tuning	1
Optimization	1
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	24
Before you finish tuning	25
Further option driven tuning	25
Options for providing application characteristics	25
Options to control optimization transformations	27
Options to assist with performance analysis	29
Options that can inhibit performance.	29

Chapter 3. Advanced optimization concepts 31

Aliasing	31
Inlining.	32
Finding the right level of inlining	32

Chapter 4. Managing code size 35

Steps for reducing code size	36
Compiler option influences on code size.	36
The -qipa compiler option	36
The -Q inlining option.	36
The -qhot compiler option	36
The -qcompact compiler option.	37
Other influences on code size	37
High activity areas	37
Computed GOTOs and CASE constructs	37
Linking and code size	38

Chapter 5. Compiler-friendly programming techniques. 41

General practices	41
Variables and pointers.	42
Arrays	42
Choosing appropriate variable sizes	42

Chapter 6. High performance libraries 43

Using the Mathematical Acceleration Subsystem libraries (MASS)	43
Using the scalar library	43
Using the vector libraries.	45
Compiling and linking a program with MASS.	49
Using the Basic Linear Algebra Subprograms (BLAS)	50
BLAS function syntax	51
Linking the libxlopt library	53

Chapter 7. Parallel programming with XL Fortran 55

Compiling your SMP code	55
The _OPENMP C preprocessor macro	55
Setting OMP and SMP run time options.	56
The XL SMP OPTS environment variable	56
OpenMP environment variables	61
Optimizing your SMP code	63
Developing and running SMP applications	64
An introduction to SMP directives.	64
Parallel region construct	64
Work-sharing constructs	65
Combined parallel work-sharing constructs.	65
Synchronization constructs	65
Other OpenMP Directives	65
Non-OpenMP SMP directives	65
Detailed descriptions of SMP directives	66
ATOMIC	66
BARRIER	68

CRITICAL / END CRITICAL	69	f_pthread_attr_getdetachstate(attr, detach)	147
DO / END DO	71	f_pthread_attr_getguardsize(attr, guardsize)	148
DO SERIAL	74	f_pthread_attr_getinheritsched(attr, inherit)	149
FLUSH	75	f_pthread_attr_getschedparam(attr, param)	149
MASTER / END MASTER	78	f_pthread_attr_getschedpolicy(attr, policy)	150
ORDERED / END ORDERED	79	f_pthread_attr_getscope(attr, scope)	150
PARALLEL / END PARALLEL	81	f_pthread_attr_getstack(attr, stackaddr, ssize)	151
PARALLEL DO / END PARALLEL DO	84	f_pthread_attr_init(attr)	151
PARALLEL SECTIONS / END PARALLEL SECTIONS	87	f_pthread_attr_setdetachstate(attr, detach)	152
PARALLEL WORKSHARE / END PARALLEL WORKSHARE	90	f_pthread_attr_setguardsize(attr, guardsize)	152
SCHEDULE	91	f_pthread_attr_setinheritsched(attr, inherit)	153
SECTIONS / END SECTIONS	94	f_pthread_attr_setschedparam(attr, param)	153
SINGLE / END SINGLE	98	f_pthread_attr_setschedpolicy(attr, policy)	154
THREADLOCAL	101	f_pthread_attr_setscope(attr, scope)	154
THREADPRIVATE	104	f_pthread_attr_setstack(attr, stackaddr, ssize)	155
WORKSHARE	108	f_pthread_attr_t	156
OpenMP directive clauses	111	f_pthread_cancel(thread)	156
Global rules for directive clauses	111	f_pthread_cleanup_pop(exec)	156
COPYIN	112	f_pthread_cleanup_push(cleanup, flag, arg)	157
COPYPRIVATE	113	f_pthread_cond_broadcast(cond)	158
DEFAULT	114	f_pthread_cond_destroy(cond)	158
IF	116	f_pthread_cond_init(cond, cattr)	159
FIRSTPRIVATE	117	f_pthread_cond_signal(cond)	159
LASTPRIVATE	118	f_pthread_cond_t	160
NUM_THREADS	119	f_pthread_cond_timedwait(cond, mutex, timeout)	160
ORDERED	120	f_pthread_cond_wait(cond, mutex)	161
PRIVATE	121	f_pthread_condattr_destroy(cattr)	161
REDUCTION	122	f_pthread_condattr_getpshared(cattr, pshared)	162
SCHEDULE	125	f_pthread_condattr_init(cattr)	162
SHARED	127	f_pthread_condattr_setpshared(cattr, pshared)	163
OpenMP execution environment, lock and timing routines	129	f_pthread_condattr_t	163
omp_destroy_lock(svar)	130	f_pthread_create(thread, attr, flag, ent, arg)	164
omp_destroy_nest_lock(nvar)	130	f_pthread_detach(thread)	165
omp_get_dynamic()	131	f_pthread_equal(thread1, thread2)	165
omp_get_max_threads()	131	f_pthread_exit(ret)	166
omp_get_nested()	132	f_pthread_getconcurrency()	166
omp_get_num_procs()	132	f_pthread_getschedparam(thread, policy, param)	167
omp_get_num_threads()	132	f_pthread_getspecific(key, arg)	167
omp_get_thread_num()	133	f_pthread_join(thread, ret)	168
omp_get_wtick()	134	f_pthread_key_create(key, dtr)	168
omp_get_wtime()	135	f_pthread_key_delete(key)	169
omp_in_parallel()	135	f_pthread_key_t	170
omp_init_lock(svar)	136	f_pthread_kill(thread, sig)	170
omp_init_nest_lock(nvar)	137	f_pthread_mutex_destroy(mutex)	170
omp_set_dynamic(enable_expr)	138	f_pthread_mutex_init(mutex, mattr)	171
omp_set_lock(svar)	138	f_pthread_mutex_lock(mutex)	171
omp_set_nested(enable_expr)	139	f_pthread_mutex_t	172
omp_set_nest_lock(nvar)	140	f_pthread_mutex_trylock(mutex)	172
omp_set_num_threads(number_of_threads_expr)	141	f_pthread_mutex_unlock(mutex)	172
omp_test_lock(svar)	141	f_pthread_mutexattr_destroy(mattr)	173
omp_test_nest_lock(nvar)	142	f_pthread_mutexattr_getpshared(mattr, pshared)	173
omp_unset_lock(svar)	142	f_pthread_mutexattr_gettype(mattr, type)	174
omp_unset_nest_lock(nvar)	143	f_pthread_mutexattr_init(mattr)	175
Pthreads library module	144	f_pthread_mutexattr_setpshared(mattr, pshared)	175
Pthreads data structures, functions, and subroutines	144	f_pthread_mutexattr_settype(mattr, type)	176
f_maketime(delay)	147	f_pthread_mutexattr_t	177
f_pthread_attr_destroy(attr)	147	f_pthread_once(once, initr)	177
		f_pthread_once_t	177
		f_pthread_rwlock_destroy(rwlock)	177
		f_pthread_rwlock_init(rwlock, rwattr)	178

f_thread_rwlock_rdlock(rwlock)	178
f_thread_rwlock_t	179
f_thread_rwlock_tryrdlock(rwlock)	179
f_thread_rwlock_trywrlock(rwlock)	180
f_thread_rwlock_unlock(rwlock)	180
f_thread_rwlock_wrlock(rwlock)	181
f_thread_rwlockattr_destroy(rwattr)	181
f_thread_rwlockattr_getpshared(rwattr, pshared)	182
f_thread_rwlockattr_init(rwattr)	182
f_thread_rwlockattr_setpshared(rwattr, pshared)	183
f_thread_rwlockattr_t	183
f_thread_self()	184
f_thread_setcancelstate(state, oldstate)	184
f_thread_setcanceltype(type, oldtype)	185
f_thread_setconcurrency(new_level)	185
f_thread_setschedparam(thread, policy, param)	186
f_thread_setspecific(key, arg)	186
f_thread_t	187
f_thread_testcancel()	187
f_sched_param	187
f_sched_yield()	188
f_timespec	188

Chapter 8. Interlanguage calls 189

Conventions for XL Fortran external names	189
Mixed-language input and output	190
Mixing Fortran and C++	191
Making calls to C functions work.	192
Passing data from one language to another	193
Passing arguments between languages	193
Passing global variables between languages	194
Passing character types between languages	194
Passing arrays between languages	195
Passing pointers between languages	196
Passing arguments by reference or by value	197
Passing complex values to/from gcc.	199
Returning values from Fortran functions	200
Arguments with the OPTIONAL attribute	200
Assembler-level subroutine linkage conventions	200
The stack	202
The Link Area and Minimum Stack Frame	204
The input parameter area	205
The register save area	205
The local stack area	205
The output parameter area	205
Linkage convention for argument passing	206
Argument passing rules (by value)	207
Order of arguments in argument list	209
Linkage convention for function calls	209
Pointers to functions	210
Function values	210
The Stack floor	211
Stack overflow	211
Prolog and epilog	211
Traceback.	211

Chapter 9. Implementation details of XL Fortran Input/Output (I/O). 213

Implementation details of file formats	213
File names	214
Preconnected and Implicitly Connected Files	214
File positioning.	215
I/O Redirection	216
How XL Fortran I/O interacts with pipes, special files, and links	216
Default record lengths	217
File permissions	217
Selecting error messages and recovery actions	217
Flushing I/O buffers	218
Choosing locations and names for Input/Output files	218
Naming files that are connected with no explicit name	218
Naming scratch files	219
Asynchronous I/O	220
Execution of an asynchronous data transfer operation.	220
Usage	220
Performance.	223
Compiler-generated temporary I/O items	223
Error handling	224
XL Fortran thread-safe I/O library	224
Use of I/O statements in signal handlers	227
Asynchronous thread cancellation	227

Chapter 10. Implementation details of XL Fortran floating-point processing . 229

IEEE Floating-point overview	229
Compiling for strict IEEE conformance	229
IEEE Single- and double-precision values	230
IEEE Extended-precision values	230
Infinities and NaNs	230
Exception-handling model	231
Hardware-specific floating-point overview.	232
Single- and double-precision values	232
Extended-precision values	233
How XL Fortran rounds floating-point calculations	234
Selecting the rounding mode	234
Minimizing rounding errors	236
Minimizing overall rounding	236
Delaying rounding until run time	236
Ensuring that the rounding mode is consistent	236
Duplicating the floating-point results of other systems	237
Maximizing floating-point performance	237
Detecting and trapping floating-point exceptions	238
Compiler features for trapping floating-point exceptions	238
Installing an exception handler	239
Producing a core file	240
Controlling the floating-point status and control register	240
xlf_fp_util Procedures	241
fpgets and fpsets subroutines	241
Sample programs for exception handling	243
Causing exceptions for particular variables	243
Minimizing the performance impact of floating-point exception trapping	243

Chapter 11. Porting programs to XL Fortran	
Fortran	245
Outline of the porting process	245
Portability of directives	245
Common industry extensions that XL Fortran supports	246
Mixing data types in statements	247
Date and time routines	247
Other libc routines	247
Changing the default sizes of data types	247
Name conflicts between your procedures and XL Fortran intrinsic procedures	247
Reproducing results from other systems	248
Finding nonstandard extensions	248

Appendix. Sample Fortran programs	249
Example 1 - XL Fortran source file	249
Execution results	249
Example 2 - valid C routine source file	250
Example 3 - valid Fortran SMP source file	252
Example 4 - invalid Fortran SMP source file	252
Programming examples using the Pthreads library module	253
Notices	255
Trademarks and service marks	257
Index	259

About this document

This document is part of the IBM® XL Fortran Advanced Edition for Linux®, V11.1 documentation 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 document

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

How to use this document

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 documentation suite.

This guide does not include information on the following topics, which are covered in other documents:

- 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: consult your Linux-specific distribution's man page help and documentation.

How this document 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 31, Chapter 4, "Managing code size," on page 35, 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 41
 - Chapter 6, “High performance libraries,” on page 43
 - Chapter 7, “Parallel programming with XL Fortran,” on page 55
 - Chapter 8, “Interlanguage calls,” on page 189
- 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 213
 - Chapter 10, “Implementation details of XL Fortran floating-point processing,” on page 229
 - Chapter 11, “Porting programs to XL Fortran,” on page 245

Conventions and terminology used in this document

Typographical conventions

The following table explains the typographical conventions used in this document.

Table 1. *Typographical conventions*

Typeface	Indicates	Example
<i>italics</i>	Parameters or variables whose actual names or values are to be supplied by the user. Italics are also used to introduce new terms.	The maximum length of the <i>trigger_constant</i> in fixed source form is 4 for directives that are continued on one or more lines.
<u>underlining</u>	The default setting of a parameter of a compiler option or directive.	nomaf <u>maf</u>
monospace	Examples of program code, command strings, or user-defined names.	Also, specify the following runtime options before running the program, with a command similar to the following: export XLF RTEOPTS="err_recovery=no:langlvl=90std"
UPPERCASE bold	Fortran programming keywords, statements, directives, and intrinsic procedures.	The ASSERT directive applies only to the DO loop immediately following the directive, and not to any nested DO loops.
lowercase bold	Lowercase programming keywords and library functions, compiler intrinsic procedures, file and directory names, examples of program code, command strings, or user-defined names.	If you specify -O3 , the compiler assumes -qhot=level=0 . To prevent all HOT optimizations with -O3 , you must specify -qnohot .

Syntax diagrams

Throughout this document, 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 \blacktriangleright — symbol indicates the beginning of a command, directive, or statement.

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

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

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

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

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

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

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

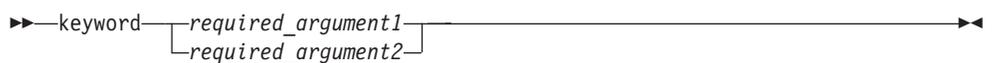


- Optional items are shown below the main path:



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

- If you can choose from two or more items, they are shown vertically, in a stack. If you *must* choose one of the items, one item of the stack is shown on the main path.



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



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



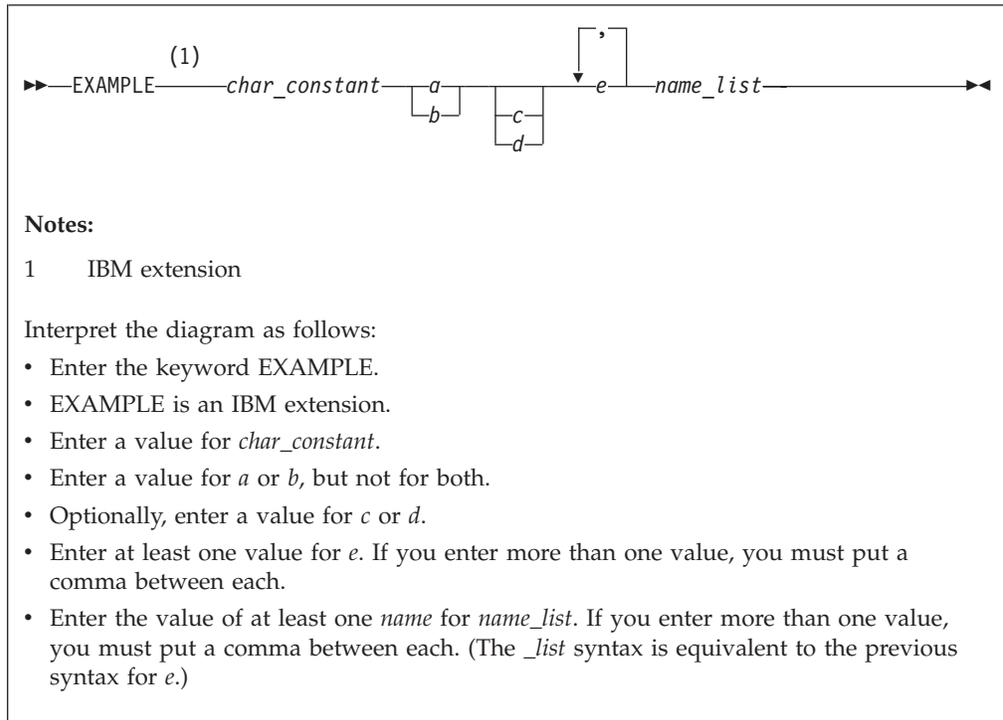
- The item that is the default is shown above the main path.



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

Sample syntax diagram

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



Examples

The examples in this document are coded in a simple style that does not try to conserve storage, check for errors, achieve fast performance, or demonstrate recommended practice.

The examples in this document use the `xlf90`, `xlf90_r`, `xlf95`, `xlf95_r`, `xlf`, `xlf_r`, `xlf2003`, `xlf2003_r`, `f2003`, `f77`, `fort77`, `f90`, and `f95` compiler invocation commands interchangeably. For more substantial source files, one of these commands may be more suitable than the others, as explained in *Compiling programs* in the *XL Fortran Compiler Reference*.

Some sample programs from this document and some other programs that illustrate ideas presented in this document are in the directory `/opt/ibmcmp/xlf/11.1/samples`.

Notes on path names

The path names shown in this document assume the default installation path for the XL Fortran compiler. By default, XL Fortran will be installed in the following directory on the selected disk: `/opt/ibmcmp/xlf/11.1/`

You can select a different destination (*relocation-path*) for the compiler. If you choose a different path, the compiler will be installed in the following directory: *relocation-path*/`/opt/ibmcmp/xlf/11.1/`

Notes on the terminology used

Some of the terminology in this document 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 information on documentation related to XL Fortran:

- “IBM XL Fortran publications”
- “Standards and specifications documents” on page xii
- “Other IBM publications” on page xiii

IBM XL Fortran publications

XL Fortran provides product documentation in the following formats:

- README files
README files contain late-breaking information, including changes and corrections to the product documentation. 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 *XL Fortran 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 *XL Fortran Installation Guide*. The information center is also viewable on the Web at <http://publib.boulder.ibm.com/infocenter/lxphelp/v9v111/index.jsp>.
- PDF documents
PDF documents are located by default in the `/opt/ibmcmp/xlf/11.1/doc/LANG/pdf/` directory, where *LANG* is one of `en_US` or `ja_JP`. The PDF files are also available on the Web at <http://www.ibm.com/software/awdtools/fortran/xlfortran/library>.

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

Table 2. XL Fortran PDF files

Document title	PDF file name	Description
IBM XL Fortran Advanced Edition for Linux, V11.1 Installation Guide, GC23-5896-00	install.pdf	Contains information for installing XL Fortran and configuring your environment for basic compilation and program execution.
Getting Started with IBM XL Fortran Advanced Edition for Linux, V11.1, GC23-5897-00	getstart.pdf	Contains an introduction to the XL Fortran product, with information on setting up and configuring your environment, compiling and linking programs, and troubleshooting compilation errors.
IBM XL Fortran Advanced Edition for Linux, V11.1 Compiler Reference, SC23-5895-00	cr.pdf	Contains information about the various compiler options and environment variables.
IBM XL Fortran Advanced Edition for Linux, V11.1 Language Reference, SC23-5894-00	lr.pdf	Contains information about the Fortran programming language as supported by IBM, including language extensions for portability and conformance to non-proprietary standards, compiler directives and intrinsic procedures.
IBM XL Fortran Advanced Edition for Linux, V11.1 Optimization and Programming Guide, SC23-5898-00	opg.pdf	Contains information on advanced programming topics, such as application porting, interlanguage calls, floating-point operations, input/output, application optimization and parallelization, and the XL Fortran high-performance libraries.

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

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

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

Standards and specifications documents

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 document.

- *American National Standard Programming Language FORTRAN, ANSI X3.9-1978.*
- *American National Standard Programming Language Fortran 90, ANSI X3.198-1992.*
- *ANSI/IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-1985.*
- *Federal (USA) Information Processing Standards Publication Fortran, FIPS PUB 69-1.*
- *Information technology - Programming languages - Fortran, ISO/IEC 1539-1:1991 (E).*
- *Information technology - Programming languages - Fortran - Part 1: Base language, ISO/IEC 1539-1:1997. (This document uses its informal name, Fortran 95.)*
- *Information technology - Programming languages - Fortran - Part 1: Base language, ISO/IEC 1539-1:2004. (This document 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.*
- *OpenMP Application Program Interface Version 2.5, available at <http://www.openmp.org>*

Other IBM publications

- *ESSL for Linux on POWER V4.2 Guide and Reference, SA22-7904, available at <http://publib.boulder.ibm.com/infocenter/clresctr/index.jsp>*

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 technical support FAQs and other support documents.

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 document or any other XL Fortran documentation, send your comments by e-mail to compinfo@ca.ibm.com.

Be sure to include the name of the document, the part number of the document, 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 for the Linux operating system 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 section 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 3. Basic optimizations

Optimization level	Additional options implied by default	Complementary options	Other options with possible benefits
-O0	None	-qarch	-g
-O2	-qmaxmem=8192	-qarch -qtune	-qmaxmem=-1 -qhot=level=0

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 and bad casting.

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 section 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

After applying basic optimizations and successfully compiling and executing your application, you can apply more powerful optimization tools. 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. 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 4 on page 4 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 4. Advanced optimizations

Optimization Level	Additional options implied	Complementary options	Options with possible benefits
-O3	-qnostrict -qmaxmem=-1 -qhot=level=0	-qarch -qtune	-qpdf

Table 4. Advanced optimizations (continued)

-O4	-qnostrict -qmaxmem=-1 -qhot -qipa -qarch=auto -qtune=auto -qcache=auto	-qarch -qtune -qcache	-qpdf -qsmp=auto
-O5	All of -O4 -qipa=level=2	-qarch -qtune -qcache	-qpdf -qsmp=auto

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 **-qnostrict**, 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 **-qstrict**. Compiling with **-qstrict** is necessary if you require the same absolute precision in floating-point computational accuracy as you get with **-O0**, **-O2**, or **-qnoopt** results. The **-qstrict** compiler option 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 **-qstrict** will help assure accurate results. Without **-qstrict**, the difference in computation for any one source-level operation is very small in comparison to basic optimization. Though a small difference can compound 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=arraypad` to enable array padding

For more information on `-qhot`, see High-order transformation (HOT).

Optimizing at level 4

Benefits at level 4

- Propagation of global and parameter 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.

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 Interprocedural analysis (IPA) 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.

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.

Optimizing at level 5

Benefits at level 5

- Most aggressive optimizations available
- Makes full use of loop optimizations and IPA

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

Table 5. Specialized optimization techniques

Technique	Benefit
HOT	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.
IPA	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.
PDF	Targets the code paths your application executes most frequently for optimization.

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.

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=arraypad** 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.

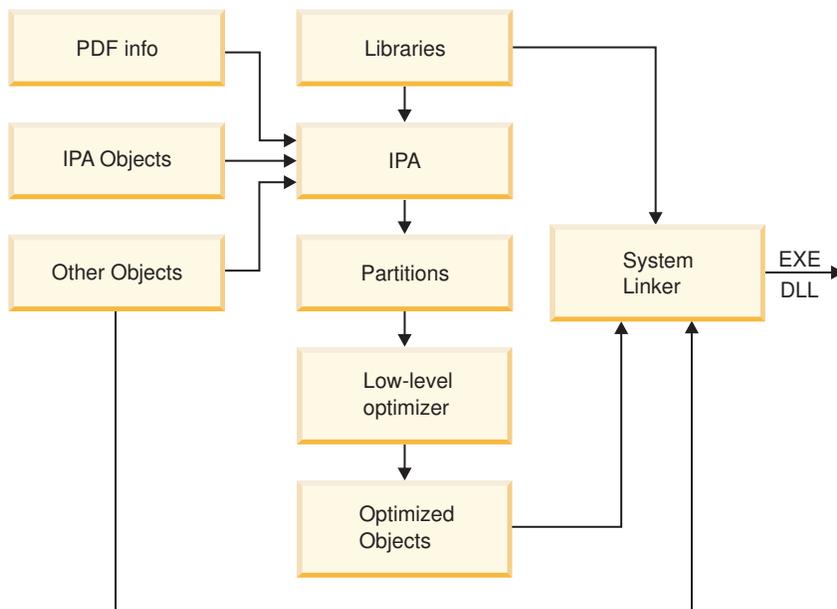


Figure 1. IPA at the link step

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.

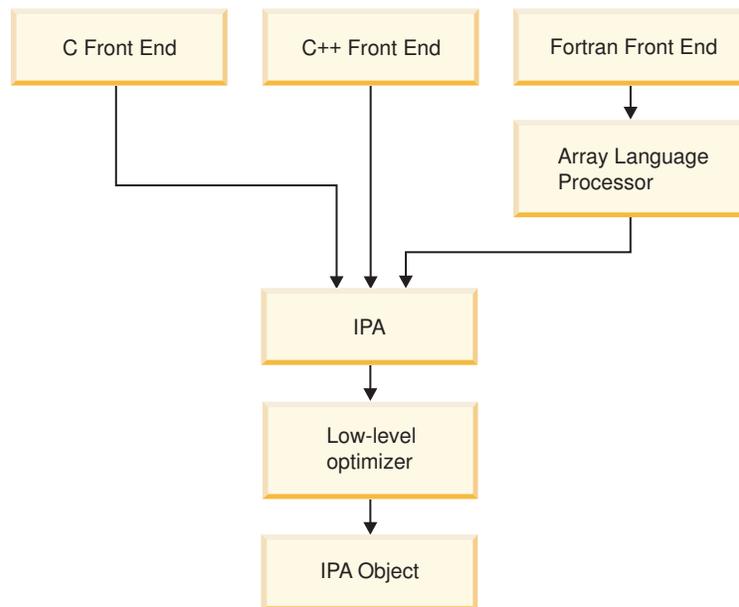


Figure 2. IPA at the compile step

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 6. The levels of IPA

IPA Level	Behaviors
-----------	-----------

Table 6. The levels of IPA (continued)

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: <ul style="list-style-type: none"> - Value numbering - Code propagation and simplification - Code motion, into conditions and out of loops - Redundancy elimination techniques

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.

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 **-pdf2** 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.

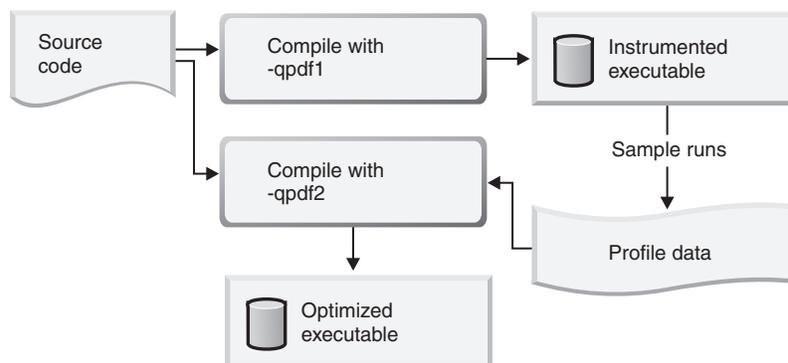


Figure 3. Profile-directed feedback

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. While PDF is recommended at **-O4** and higher, you can specify **-qpdf** as early in the optimization process as **-O2** but will not necessarily achieve optimal results.

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:

```
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:

```
xlf -c -O3 -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.

```
xlf -O3 -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.

```
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**.

```
xlf -c -O3 -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” on page 16 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.
- qipa=list** Instructs the compiler to emit an object listing that provides information for IPA optimization.

You can also use the **SNAPSHOT** 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:

```
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 on page 18 **Original code**: 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 **dbx debugger listing**: Represents a listing of the optimized source as shown in the dbx debugger. Note the unrolled loop and the consolidation of values assigned by the $x + y$ expression.

Figure 6 on page 20 **Stepping through optimized source**: Shows an example of stepping through the optimized source using the dbx 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.

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

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

Figure 4. Original code

```

(dbx) list
 1
 2
 3     1|      REAL*4 FUNCTION foo (x, y)
 4     1|      @CSE2 = x
 5     5|      @CSE1 = y
 6     5|      #2 = _x1fBeginIO(6,257,#1,1024,NULL,0,NULL)
 7     5|      @CSE0 = @CSE2 + @CSE1
 8     5|      #3 = @CSE0
 9     5|      CALL _x1fWriteLDReal(%VAL(#2),#3,4,4)
10     5|      #4 = @CSE0
11     5|      CALL _x1fWriteLDReal(%VAL(#2),#4,4,4)
12     5|      _x1fEndIO(%VAL(#2))
13     5|      #2 = _x1fBeginIO(6,257,#1,1024,NULL,0,NULL)
14     5|      #3 = @CSE0
15     5|      CALL _x1fWriteLDReal(%VAL(#2),#3,4,4)
16     5|      #4 = @CSE0
17     5|      CALL _x1fWriteLDReal(%VAL(#2),#4,4,4)
18     5|      _x1fEndIO(%VAL(#2))
19     5|      #2 = _x1fBeginIO(6,257,#1,1024,NULL,0,NULL)
20     5|      #3 = @CSE0
21     5|      CALL _x1fWriteLDReal(%VAL(#2),#3,4,4)
22     5|      #4 = @CSE0
23     5|      CALL _x1fWriteLDReal(%VAL(#2),#4,4,4)
24     5|      _x1fEndIO(%VAL(#2))
25     5|      #2 = _x1fBeginIO(6,257,#1,1024,NULL,0,NULL)
26     5|      #3 = @CSE0
27     5|      CALL _x1fWriteLDReal(%VAL(#2),#3,4,4)
28     5|      #4 = @CSE0
29     5|      CALL _x1fWriteLDReal(%VAL(#2),#4,4,4)
30     5|      _x1fEndIO(%VAL(#2))
31     8|      RETURN
32     8|      END FUNCTION foo
33
34
35     10|      PROGRAM main ()
36     11|      T_3 = 3.00000000E+00
37     11|      T_4 = 4.00000000E+00
38     11|      CALL foo(T_3,T_4)
39     12|      CALL _x1fExit(0)
40     12|      CALL _trap(3)
41     12|      END PROGRAM main

```

Figure 5. dbx debugger listing

```

(dbx) stop at 17
[1] stop at "myprogram.o.rptdbg":17
(dbx) cont
7.000000000 7.000000000
[1] stopped in foo at line 17 in file "myprogram.o.rptdbg"
17          CALL _xlfWriteLDReal(%VAL(#2),#4,4,4)
(dbx) step
7.000000000 7.000000000
stopped in foo at line 18 in file "myprogram.o.rptdbg"
18          _xlfEndIO(%VAL(#2))
(dbx) step
stopped in foo at line 20 in file "myprogram.o.rptdbg"
20          #3 = @CSE0
(dbx) step
stopped in foo at line 22 in file "myprogram.o.rptdbg"
22          #4 = @CSE0
(dbx) step
stopped in foo at line 23 in file "myprogram.o.rptdbg"
23          CALL _xlfWriteLDReal(%VAL(#2),#4,4,4)
(dbx) step
7.000000000 7.000000000
stopped in foo at line 24 in file "myprogram.o.rptdbg"
24          _xlfEndIO(%VAL(#2))
(dbx) step
stopped in foo at line 26 in file "myprogram.o.rptdbg"
26          #3 = @CSE0
(dbx) cont
7.000000000 7.000000000

execution completed

```

Figure 6. Stepping through optimized source

Getting more 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 tuning alternatives. See the following sections for details:

- Chapter 2, “Tuning XL compiler applications,” on page 21
- Chapter 3, “Advanced optimization concepts,” on page 31
- “Optimizing your SMP code” on page 63

Beyond performance: effective programming techniques

Applications that perform well begin with applications that are written well. This section contains information on how to write 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 41
- Chapter 4, “Managing code size,” on page 35
- Chapter 7, “Parallel programming with XL Fortran,” on page 55
- Chapter 8, “Interlanguage calls,” on page 189

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 Chapter 1, “Optimizing your applications,” on page 1, and discusses the next steps in increasing the performance of your application:

- Tuning for your target architecture
- Further option driven tuning

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 7. Options for targeting your architecture

Option	Behavior
-q32	Generates code for a 32-bit addressing model (32-bit execution mode).
-q64	Generates code for a 64-bit addressing model (64-bit execution mode).
-qarch	Selects a family of processor architectures, or a specific architecture that the compiler will generate machine instructions for. If you specify multiple architecture settings, only the last architecture is considered valid.
-qtune	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.
-qcache	Defines a specific cache or memory geometry. Selecting a predefined optimization level like -O2 sets default values for -qcache suboptions.
-qipa=clonearch	Instructs the compiler to generate duplicate subprograms with each tuned to a particular architecture.

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 Chapter 1, “Optimizing your applications,” on page 1 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.

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 8. Instruction group support by architecture

-qarch suboption	Available				Instructions			
	PowerPC	Graphics	Sqrt	64-bit	PWR3	PWR4	PWR5	VMX
ppc family	X							
ppcgr family	X	X						
ppc64 family	X			X				
ppc64gr	X	X		X				
ppc64grsq	X	X	X	X				
rs64b	X	X	X	X				
rs64c	X	X	X	X				
pwr3 chip and family	X	X	X	X	X			
pwr4 chip and family	X	X	X	X	X	X		
pwr5 chip and family	X	X	X	X	X	X	X	
pwr5x chip	X	X	X	X	X	X	X	
ppc64v family	X	X	X	X	X	X		X
ppc970 chip	X	X	X	X	X	X		X
-qarch suboption	PowerPC	Graphics	Sqrt	64-bit	PWR3	PWR4	PWR5	VMX

Using the default value for **-qarch** represents the broadest possible range of machines that the compiler supports. For example, the defaults to a setting of `ppc64grsq`. If you know that your code will only execute on POWER5 machines, avoid the default **-qarch** setting and choose at least `PWR5`, instead.

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 Linux, the setting is PPC64GRSQ. 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 PWR5 machine 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 **-qcache**

The **-qcache** 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 **-qcache** option is only effective if you understand the cache characteristics of the execution environment of your application. Before using **-qcache**, 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 **-qcache**, and allow the compiler to use default cache settings.

If you do not specify **-qcache**, the compiler makes cache assumptions based on your **-qarch** and **-qtune** settings. If you compile with the **-qcache=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 **-qcache**, also specify **-qhot**, or an option such as **-O4** that implies **-qhot**. The optimizations that **-qhot** performs are designed to take advantage of your **-qcache** 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 **-qcache**, specify **-qhot** as well, which can take advantage of your cache settings.

Further option driven tuning

You can use the options in this section 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 the Advanced optimization concepts section.

- 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.
- qfltrap** 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.

- 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.
- qlibessl** 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. The High performance libraries section contains more information on ESSL.
- 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.
- qonetrrip** 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 `xlfr`, 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.

- qenablevmx** Allows you to take advantage of the VMX capabilities of chips such as the PPC970. This is the default setting.
- 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.
- 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.
- qsmp** 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. Do not use this option unless your application requires strict adherence as **-qstrict** 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 the *Profile-directed feedback* section.
- qtbtable**
Limits the amount of debugging traceback information in object files, which reduces the size of the program. Use **-qtbtable=full** if you intend to analyze your application with a 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.

Consult the *XL Fortran Compiler Reference* or the relevant options in this section for complete descriptions of the following options.

Table 9. Options that can reduce performance

<code>-qalias=nostd</code>	<code>-qfloat=nosqrt</code>	<code>-qstacktemp=[value other than 0 or -1]</code>	<code>-qunwind</code>
<code>-qcompact</code>	<code>-qfloat=nostrictmaf</code>	<code>-qstrict</code>	<code>-qzerosize</code>
<code>-qnoenablevmx</code>	<code>-qnoprefetch</code>	<code>-qstrictieemod</code>	
<code>-qfloat=norelax</code>	<code>-Q!</code>	<code>-qstrict_induction</code>	
<code>-qfloat=rrm</code>	<code>-qsmallstack</code>	<code>-qnounroll</code>	

Chapter 3. Advanced optimization concepts

After you apply command-line optimizations and tuning appropriate to your application and the constraints of your development cycle, this section can provide you with further information on opportunities to improve the performance of your application. See the following concepts for more information:

- “Aliasing”
- “Inlining” on page 32

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 25.

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.

```
# Let the compiler decide (relatively cautiously) what to inline.
xlf95 -O3 -qipa=inline inline.f
```

```
# Encourage the compiler to inline particular subprograms.
xlf95 -O3 -qipa=inline=called_100_times,called_1000_times inline.f
```

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. 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 an **EQUIVALENCE** 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

This section outlines some 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 **-qcompact** that can help reduce code size. See *Compiler option influences on code size* for more information.

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. The following sections detail other compiler options that can influence the size of your code.

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.

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:

```
GOTO (10,10,10,10,20,10,10,10,10,30,20,10,10,10,10,  
+10,20,10,20,10,20,30,30,10,10,10,10,10,10,20,10,10,...  
+10,20,30,10,10,10,30,10,10,10,10,10,10,20,10,30) IA(I)  
  
GOTO 10  
30 CONTINUE  
! ...  
GOTO 10  
20 CONTINUE  
! ...  
10 CONTINUE
```

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.

```
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.

Linking and code size

Dynamic linking

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

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 can be as important to the performance of your application as the compilation options that you specify. This section contains suggestions on writing code with the optimizer and portability in mind and contains the following:

- “General practices”
- “Variables and pointers” on page 42
- “Arrays” on page 42
- “Choosing appropriate variable sizes” on page 42

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.

Using command invocations like **xlF90** and **xlF95** will enhance standards conformance and code portability.

Specifying **-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**, and **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

Obey all aliasing rules. Avoid specifying **-qalias=nostd**. For more information on aliasing and how it can affect performance, see “Aliasing” on page 31.

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

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 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)” on page 50.

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 45. 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 49 describes how to compile and link a program that uses the MASS libraries, and how to selectively use the MASS scalar library functions in concert with the regular system library scalar functions.

Note: On Linux, 32-bit and 64-bit objects cannot be combined in the same library, so two versions of the scalar and vector libraries are shipped with the compiler: `libmass.a` and `libmassv.a` for 32-bit applications and `libmass_64.a` and `libmassv_64.a` for 64-bit applications.

Using the scalar library

The MASS scalar libraries `libmass.a` (32-bit) and `libmass_64.a` (64-bit) contain an accelerated set of frequently used math intrinsic functions that provide improved performance over the corresponding standard system library functions. These functions are available when you compile programs with any of the following options:

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

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` (or the 64-bit version, `libmass_64.a`) with your application. For instructions, see “Compiling and linking a program with MASS” on page 49
2. All the MASS scalar routines, except those listed in 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 3, include `mass.include` in your source file.
3. Include `mass.include` in your source file for the following functions:
 - `acosf`, `acosh`, `acoshf`, `asinf`, `asinh`, `asinhf`, `atan2f`, `atanf`, `atanh`, `atanhf`, `cbrt`, `cbrtf`, `copysign`, `copysignf`, `cosf`, `coshf`, `cosisin`, `erff`, `erfcf`, `expf`, `expm1f`, `hypot`, `hypotf`, `lgammaf`, `logf`, `log10f`, `log1pf`, `rsqrt`, `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 10.

Table 10. MASS scalar functions

Double-precision function	Single-precision function	Description
<code>acos</code>	<code>acosf</code>	Returns the arccosine of x
<code>acosh</code>	<code>acoshf</code>	Returns the hyperbolic arccosine of x
	<code>anint</code>	Returns the rounded integer value of x
<code>asin</code>	<code>asinf</code>	Returns the arcsine of x
<code>asinh</code>	<code>asinhf</code>	Returns the hyperbolic arcsine of x
<code>atan2</code>	<code>atan2f</code>	Returns the arctangent of x/y
<code>atan</code>	<code>atanf</code>	Returns the arctangent of x
<code>atanh</code>	<code>atanhf</code>	Returns the hyperbolic arctangent of x
<code>cbrt</code>	<code>cbrtf</code>	Returns the cube root of x
<code>copysign</code>	<code>copysignf</code>	Returns x with the sign of y
<code>cos</code>	<code>cosf</code>	Returns the cosine of x
<code>cosh</code>	<code>coshf</code>	Returns the hyperbolic cosine of x
<code>cosisin</code>		Returns a complex number with the real part the cosine of x and the imaginary part the sine of x .
<code>dnint</code>		Returns the nearest integer to x (as a double)
<code>erf</code>	<code>erff</code>	Returns the error function of x
<code>erfc</code>	<code>erfcf</code>	Returns the complementary error function of x
<code>exp</code>	<code>expf</code>	Returns the exponential function of x
<code>expm1</code>	<code>expm1f</code>	Returns (the exponential function of x) $- 1$
<code>hypot</code>	<code>hypotf</code>	Returns the square root of $x^2 + y^2$

Table 10. MASS scalar functions (continued)

Double-precision function	Single-precision function	Description
lgamma	lgammaf	Returns the natural logarithm of the absolute value of the Gamma function of x
log	logf	Returns the natural logarithm of x
log10	log10f	Returns the base 10 logarithm of x
log1p	log1pf	Returns the natural logarithm of $(x + 1)$
rsqrt		Returns the reciprocal of the square root of x
sin	sinf	Returns the sine of x
sincos		Sets $*s$ to the sine of x and $*c$ to the cosine of x
sinh	sinhf	Returns the hyperbolic sine of x
sqrt		Returns the square root of x
tan	tanf	Returns the tangent of x
tanh	tanhf	Returns the hyperbolic tangent of x
$x^{**}y$		Returns x raised to the power y

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

```
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).

Using the vector libraries

When you compile programs with any of the following options:

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

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`, `vdhint`, `vdint`, `vsincos`, `vssincos`, `vcosisin`, `vscosisin`, `vqdrft`, `vsqdrft`, `vrqdrft`, `vsrqdrft`, `vpopcnt4`, and `vpopcnt8`). 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 49.)

libmassvp4.a

Contains functions that have been tuned for the POWER4 architecture. If you are using a PPC970 machine, this library is the recommended choice.

libmassvp5.a

Contains functions that have been tuned for the POWER5 architecture.

libmassvp6.a

Contains functions that have been tuned for the POWER6™ architecture.

On Linux, 32-bit and 64-bit objects must not be mixed in a single library, so a separate 64-bit version of each vector library is provided: `libmassvp4_64.a`, `libmassvp5_64.a`, and `libmassvp6_64.a`

The single-precision and double-precision floating-point functions contained in the vector libraries are summarized in Table 11 on page 47. The integer functions contained in the vector libraries are summarized in Table 12 on page 48.

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:

```
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 $i=1,\dots,500$.

The functions `vdiv`, `vsincos`, `vpow`, and `vatan2` (and their single-precision versions, `vsdiv`, `vssincos`, `vspow`, and `vsatan2`) take four parameters. The functions `vdiv`, `vpow`, and `vatan2` take the parameters (*z,x,y,n*). The function `vdiv` outputs a vector *z* whose elements are $x(i)/y(i)$, where $i=1,\dots,n$. The function `vpow` outputs a vector *z* whose elements are $x(i)^{y(i)}$, where $i=1,\dots,n$. The function `vatan2` outputs a vector *z* whose elements are $\text{atan}(x(i)/y(i))$, where $i=1,\dots,n$. The function `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.

Table 11

Table 11. MASS floating-point vector library functions

Double-precision function	Single-precision function	Arguments	Description
vacos	vsacos	(y, x, n)	Sets y(i) to the arc cosine of x(i), for i=1,..,n
vacosh	vsacosh	(y, x, n)	Sets y(i) to the hyperbolic arc cosine of x(i), for i=1,..,n
vasin	vsasin	(y, x, n)	Sets y(i) to the arc sine of x(i), for i=1,..,n
vasinh	vsasinh	(y, x, n)	Sets y(i) to the arc hyperbolic sine of x(i), for i=1,..,n
vatan2	vsatan2	(z, x, y, n)	Sets z(i) to the arc tangent of x(i)/y(i), for i=1,..,n
vatanh	vsatanh	(y, x, n)	Sets y(i) to the arc hyperbolic tangent of x(i), for i=1,..,n
vcbrt	vscbrt	(y, x, n)	Sets y(i) to the cube root of x(i), for i=1,..,n
vcos	vscos	(y, x, n)	Sets y(i) to the cosine of x(i), for i=1,..,n
vcosh	vscosh	(y, x, n)	Sets y(i) to the hyperbolic cosine of x(i), for i=1,..,n
vcosisin	vscosisin	(y, x, n)	Sets the real part of y(i) to the cosine of x(i) and the imaginary part of y(i) to the sine of x(i), for i=1,..,n
vdint		(y, x, n)	Sets y(i) to the integer truncation of x(i), for i=1,..,n
vdiv	vsdiv	(z, x, y, n)	Sets z(i) to x(i)/y(i), for i=1,..,n
vdnint		(y, x, n)	Sets y(i) to the nearest integer to x(i), for i=1,..,n
vexp	vsexp	(y, x, n)	Sets y(i) to the exponential function of x(i), for i=1,..,n
vexpm1	vsexpm1	(y, x, n)	Sets y(i) to (the exponential function of x(i))-1, for i=1,..,n
vlog	vslog	(y, x, n)	Sets y(i) to the natural logarithm of x(i), for i=1,..,n
vlog10	vslog10	(y, x, n)	Sets y(i) to the base-10 logarithm of x(i), for i=1,..,n
vlog1p	vslog1p	(y, x, n)	Sets y(i) to the natural logarithm of (x(i)+1), for i=1,..,n
vpow	vspow	(z, x, y, n)	Sets z(i) to x(i) raised to the power y(i), for i=1,..,n
vqdrft	vsqdrft	(y, x, n)	Sets y(i) to the 4th root of x(i), for i=1,..,n
vrcbrt	vsrbrt	(y, x, n)	Sets y(i) to the reciprocal of the cube root of x(i), for i=1,..,n

Table 11. MASS floating-point vector library functions (continued)

Double-precision function	Single-precision function	Arguments	Description
vrec	vsrec	(y,x,n)	Sets y(i) to the reciprocal of x(i), for i=1,..,n
vrqdrft	vsrqdrft	(y,x,n)	Sets y(i) to the reciprocal of the 4th root of x(i), for i=1,..,n
vrsqrt	vsrsqrt	(y,x,n)	Sets y(i) to the reciprocal of the square root of x(i), for i=1,..,n
vsin	vssin	(y,x,n)	Sets y(i) to the sine of x(i), for i=1,..,n
vsincos	vssincos	(y,z,x,n)	Sets y(i) to the sine of x(i) and z(i) to the cosine of x(i), for i=1,..,n
vsinh	vssinh	(y,x,n)	Sets y(i) to the hyperbolic sine of x(i), for i=1,..,n
vsqrt	vssqrt	(y,x,n)	Sets y(i) to the square root of x(i), for i=1,..,n
vtan	vstan	(y,x,n)	Sets y(i) to the tangent of x(i), for i=1,..,n
vtanh	vstanh	(y,x,n)	Sets y(i) to the hyperbolic tangent of x(i), for i=1,..,n

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 12. MASS integer vector library functions

Function	Description	Interface
vpopcnt4	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	integer*4 function vpopcnt4 (x, n) integer*4 x(*), n
vpopcnt8	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	integer*4 function vpopcnt8 (x, n) integer*8 x(*) integer*4 n

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 vrsqrt (y, x, n)
  real*8 y(*), x(*)
  integer n      ! Sets y(i) to the reciprocal of the square root of x(i),
```

```

                ! for i=1,..,n
end subroutine vrsqrt

end interface

```

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

```

interface

subroutine vssqrt (y, x, n)
  real*4 y(*), x(*)
  integer n      ! Sets y(i) to the square root of x(i), for i=1,..,n
end subroutine vssqrt

subroutine vsrsqrt (y, x, n)
  real*4 y(*), x(*)
  integer n      ! Sets y(i) to the reciprocal of the square root of x(i),
                ! for i=1,..,n
end subroutine vsrsqrt

end interface

```

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)`). Other kinds of overlap (where input and output vectors are neither disjoint nor identical) should be avoided, since they may produce unexpected results:

- 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 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, `y(1:n)` and `x1(1:n)` must be either disjoint or identical; and `y(1:n)` and `x2(1:n)` must be either disjoint or identical.
- 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, `y1(1:n)` and `x(1:n)` must be either disjoint or identical; and `y2(1:n)` and `x(1:n)` must be either disjoint or identical. Also, the vectors `y1(1:n)` and `y2(1:n)` must be disjoint.

Consistency of MASS vector functions

All the functions in the MASS vector libraries are consistent, in the sense that a given input value will always produce the same result, regardless of its position in the vector, and regardless of the vector length.

Compiling and linking a program with MASS

To compile an application that calls the functions in the MASS libraries, specify **mass** and **massvp4**, **massvp5**, or **massvp6** (32-bit), or **mass_64** and **massvp4_64**, **massvp5_64**, or **massvp6_64** (64-bit) on the `-l` linker option.

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

```
xlf progf.f -o progf -lmass -lmassvp4
xlf progf.f -o progf -lmass_64 -lmassvp4_64 -q64
```

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 (or libmass_64.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. Use the `ar` command to extract the object files of the desired functions from libmass.a or libmass_64.a. For most functions, the object file name is the function name followed by `.s32.o` (for 32-bit mode) or `.s64.o` (for 64-bit mode).¹ For example, to extract the object file for the `tan` function in 32-bit mode, the command would be:

```
ar -x tan.s32.o libmass.a
```

2. Archive the extracted object files into another library:

```
ar -qv libfasttan.a tan.s32.o
ranlib libfasttan.a
```

3. Create the final executable using `xlf`, specifying `-lfasttan` instead of `-lmass`:

```
xlf sample.f -o sample -L dir_containing_libfasttan -lfasttan
```

This links only the `tan` function from MASS (now in `libfasttan.a`) and the remainder of the math functions from the standard system library.

Exceptions:

1. The `sin` and `cos` functions are both contained in the object files `sincos.s32.o` and `sincos.s64.o`. The `cosisin` and `sincos` functions are both contained in the object file `cosisin.s32.o`.
2. The XL C/C++ `pow` function or XL Fortran `**` (exponentiation) operator is contained in the object files `dxys32.o` and `dxys64.o`.

Note: The `cos` and `sin` functions will both be exported if either one is exported. `cosisin` and `sincos` will both be exported if either one is exported.

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” on page 51 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 (ESSL); 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 53 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
- the length of vector *y*, if 'T' or 't' is used for the *trans* parameter

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

alpha

is the scaling constant α

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

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*.

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

incx

is the stride for vector *x*. It can have any value.

beta

is the scaling constant β

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

incy

is the stride for vector *y*. It must not be zero.

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:

```
CALL SGEMM(transa, transb, l, n, m, alpha, a, lda, b, ldb, beta, c, ldc)
CALL DGEMM(transa, transb, l, n, m, alpha, a, lda, b, ldb, beta, c, ldc)
```

The parameters are as follows:

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

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 *transa* parameter
- the number of rows in matrix *a*, if 'T' or 't' is used for the *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 α

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 β

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

This section details several aspects of parallel programming.

- Compiling your SMP code
- Setting OMP and SMP runtime options
- Optimizing your SMP code
- SMP directives
- 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.

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:

- **xl_f_r**
- **xl_f90_r**
- **xl_f95_r**
- **xl_f2003_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*.

The `_OPENMP C` preprocessor macro

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 the section on 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) :: AI,BI,CI
        INTEGER(KIND=8)                :: SUMI
!$   INTEGER OMP_GET_NUM_THREADS

        COMMON/DATA/ AI,BI,CI
!$OMP THREADPRIVATE (/DATA/)

!$OMP PARALLEL
    FORALL(I=1:N,J=1:N) AI(I,J) = (I-N/2)**2+(J+N/2)
    FORALL(I=1:N,J=1:N) BI(I,J) = 3-((I/2)+(J-N/2)**2)
!$OMP MASTER
    NTHREADS=8
!$   NTHREADS=OMP_GET_NUM_THREADS()
!$OMP END MASTER
!$OMP END PARALLEL

!$OMP PARALLEL DEFAULT(PRIVATE),COPYIN(AI,BI),SHARED(NTHREADS)
!$OMP DO
    DO I=1,NTHREADS
        CALL IMAT_MUL(SUMI)
    ENDDO
!$OMP END DO
!$OMP END PARALLEL

        END

```

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

Setting OMP and SMP run time options

The following sections detail setting the necessary environment variables for writing parallel code. Please see:

- “The XLSMPOPTS environment variable”
- “OpenMP environment variables” on page 61

The XLSMPOPTS environment variable

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

```

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 *number_of_threads* partitions. The number of iterations that these partitions contain is:

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

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

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

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

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

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

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

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

static[=*n*] The iterations of a loop are divided into chunks that contain *n* iterations. Threads are assigned chunks in a "round-robin"

fashion. This is known as block cyclic scheduling. If the value of n is 1, the scheduling type is specifically referred to as cyclic scheduling.

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

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

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

Parallel execution options

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

parthds=num	Specifies the number of threads (num) to be used for parallel execution of code that you compiled with the -qsmp option. By default, this is equal to the number of online processors. There are some applications that cannot use more than some maximum number of processors. There are also some applications that can achieve performance gains if they use more threads than there are processors. This option allows you full control over the number of execution threads. The default value for num is 1 if you did not specify -qsmp . Otherwise, it is the number of online processors on the machine. For more information, see the NUM_PARTHDS intrinsic function in the <i>XL Fortran Language Reference</i> .
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 <i>XL Fortran Language Reference</i> .
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 the **-qsmp=stackcheck** is in effect, enables stack overflow checking for slave 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 the **-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 **xlsmp** 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:
`parthds=8 : schedule=dynamic=10 : spins=0 : yields=0`

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

Options to enable and control dynamic profiling

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

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

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

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

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

profilefreq=*num* Specifies the frequency with which a loop should be revisited by the dynamic profiler to determine its appropriateness for parallel or serial execution. Loops in a program can be data dependent. The loop that was chosen to execute serially with a pass of dynamic profiling may benefit from parallelization in

subsequent executions of the loop, due to different data input. Therefore, you need to examine these loops periodically to reevaluate the decision to serialize a parallel loop at run time.

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

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

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

OpenMP environment variables

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

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

OMP_DYNAMIC environment variable

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

```
▶▶—OMP_DYNAMIC=—TRUE  
FALSE————▶▶
```

If you set this environment variable to **TRUE**, the 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:

```
▶▶—OMP_NUM_THREADS=—num—————▶▶
```

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

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

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

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(Iter/N)
```

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

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

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

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

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

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.

Commonly used `-qsmp` suboptions

Suboption	Behavior
auto	Instructs the compiler to automatically generate parallel code where possible without user assistance. This option also recognizes all the SMP directives.
omp	Enforces compliance with the OpenMP API for specifying explicit parallelism.
opt	Instructs the compiler to optimize as well as parallelize. The optimization is equivalent to <code>-O2 -qhot</code> in the absence of other optimization options. The default setting of <code>-qsmp</code> is <code>-qsmp=auto:noop:opt</code> .

Commonly used **-qsmp** suboptions

Suboption	Behavior
<i>suboptions</i>	Other values for the suboption provide control over thread scheduling, nested parallelism, locking, and so on.

Developing and running SMP applications

- By default, the parallelization performed is both user-directed and automatic. Use **-qsmp=omp:noauto** 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 **xlfr**) when using **-qsmp**.
- By default, the runtime uses all available processors. Do not set the **XLSMPOPTS=PARHDS** 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 described in this section 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:

PARALLEL	END PARALLEL
-----------------	---------------------

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:

DO	END DO
SECTIONS	END SECTIONS
WORKSHARE	END WORKSHARE

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:

PARALLEL DO	END PARALLEL DO
PARALLEL SECTIONS	END PARALLEL SECTIONS
PARALLEL WORKSHARE	END PARALLEL WORKSHARE

Synchronization constructs

The following directives allow you to synchronize the execution of a parallel region by multiple threads in a team:

ATOMIC	
BARRIER	
CRITICAL	END CRITICAL
FLUSH	
ORDERED	END ORDERED
SINGLE	END SINGLE

Other OpenMP Directives

The following OpenMP directives provide additional SMP functionality:

MASTER	END MASTER
THREADPRIVATE	

Non-OpenMP SMP directives

The following directives provide additional SMP functionality:

DO SERIAL	
SCHEDULE	
THREADLOCAL	

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.

```
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 5 4

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

```
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*.

```
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 Information

- “CRITICAL / END CRITICAL” on page 69
- “PARALLEL / END PARALLEL ” on page 81
- `-qsmp` option in the *XL Fortran 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



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** 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.

```

        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
!$OMP END DO NOWAIT
        ...
!$OMP BARRIER
        C = A + B
!$OMP END PARALLEL
        END

```

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.

```

!$OMP PARALLEL DEFAULT(SHARED)

!$OMP CRITICAL
        DO I = 1, 10
            X= X + 1
!$OMP BARRIER
            Y= Y + I*I
        END DO
!$OMP END CRITICAL
!$OMP END PARALLEL

```

Related Information

- “FLUSH” on page 75
- “PARALLEL / END PARALLEL ” on page 81
- `-qsmp` option in the *XL Fortran Compiler Reference*

CRITICAL / END CRITICAL

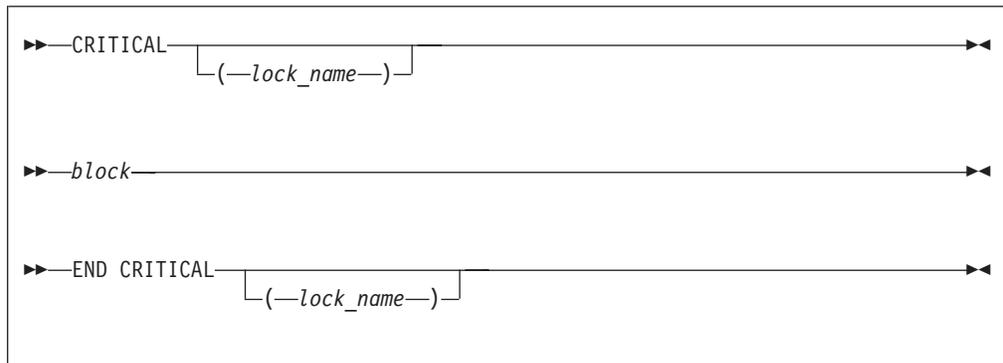
Purpose

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



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 Information

- “ATOMIC” on page 66
- “FLUSH” on page 75
- “PARALLEL / END PARALLEL ” on page 81
- *Local entity* in the *XL Fortran Language Reference*
- **-qsmp** option in the *XL Fortran Compiler Reference*

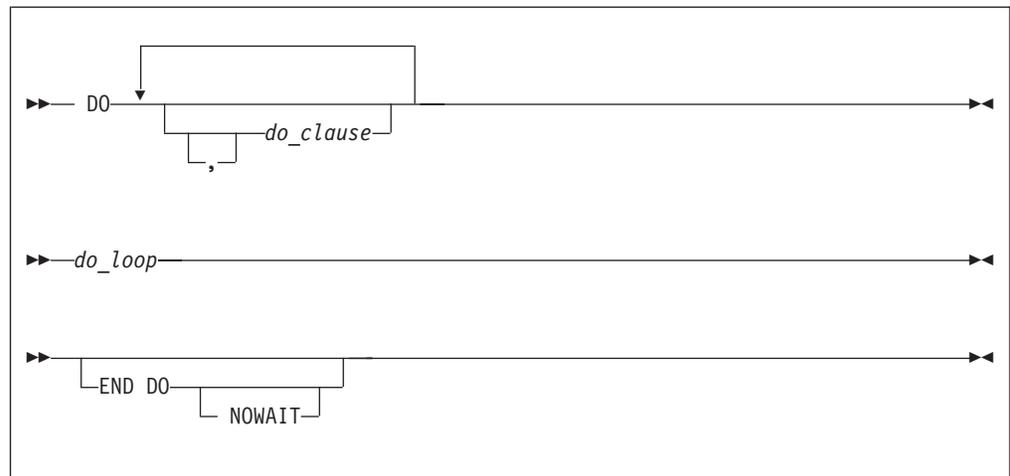
DO / END DO

Purpose

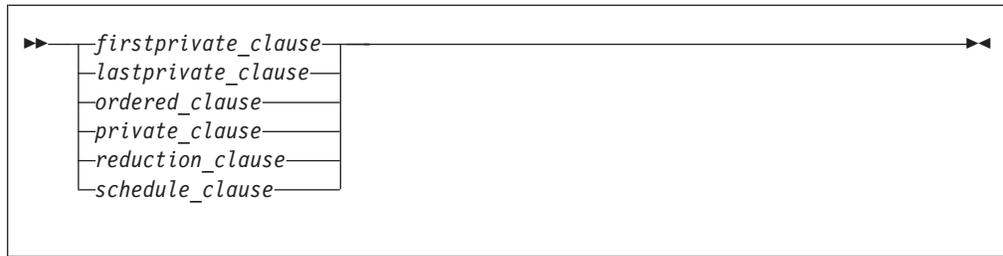
The **DO** (work-sharing) construct enables you to divide the execution of the loop among the members of the team that encounter it. The **END DO** directive enables you to indicate the end of a **DO** loop that is specified by the **DO** (work-sharing) directive.

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

Syntax



where *do_clause* is:



firstprivate_clause

See — “FIRSTPRIVATE” on page 117.

lastprivate_clause

See — “LASTPRIVATE” on page 118.

ordered_clause

See — “ORDERED” on page 120

private_clause

See — “PRIVATE” on page 121.

reduction_clause

See — “REDUCTION” on page 122

schedule_clause

See — “SCHEDULE” on page 125

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.

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.

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.

```
!$OMP PARALLEL
!$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.

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

```
!$OMP PARALLEL DEFAULT(PRIVATE), SHARED(X)
    Y = 100
!$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.

```
    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.

```

!$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 Information

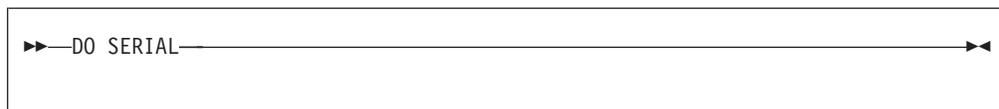
- **DO** in the *XL Fortran Language Reference*
- “DO SERIAL”
- “FLUSH” on page 75
- **INDEPENDENT** in the *XL Fortran Language Reference*
- *Loop Parallelization* in the *XL Fortran Language Reference*
- “ORDERED / END ORDERED” on page 79
- “PARALLEL / END PARALLEL ” on page 81
- “PARALLEL DO / END PARALLEL DO” on page 84
- “PARALLEL SECTIONS / END PARALLEL SECTIONS ” on page 87
- “SCHEDULE ” on page 91

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



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.

```
!$OMP PARALLEL DO PRIVATE(S,I), SHARED(A)
  DO I=1, 500
    S=0
    !SMP$ 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.

```
      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 Information

- “**DO / END DO**” on page 71
- **DO** in the *XL Fortran Language Reference*
- *Loop parallelization* in the *XL Fortran Language Reference*
- “**PARALLEL DO / END PARALLEL DO**” on page 84
- **-qdirective** option in the *XL Fortran Compiler Reference*
- **-qsmp** option in the *XL Fortran Compiler Reference*

FLUSH

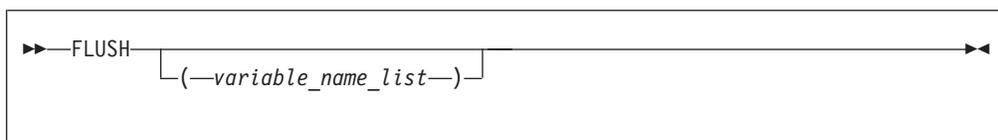
Purpose

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:

```

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
!$OMP FLUSH(INSYNC)
  INSYNC(IAM) = 1                                ! Each thread sets a flag
                                                ! once it has
!$OMP FLUSH(INSYNC)                              ! completed its work.
  DO WHILE (INSYNC(1-IAM) .eq. 0)              ! One thread waits for
                                                ! another to complete
!$OMP  FLUSH(INSYNC)                             ! its work.
  END DO

!$OMP END PARALLEL

  END PROGRAM P

  SUBROUTINE WORK                                ! Each thread does indep-
                                                ! endent calculations.
!      ...
!$OMP FLUSH                                     ! flush work variables
                                                ! before INSYNC
                                                ! is flushed.
  END SUBROUTINE WORK

```

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

```

        FUNCTION F()
        INTEGER, AUTOMATIC :: i
!$OMP FLUSH(I)
        END FUNCTION F

```

MASTER / END MASTER

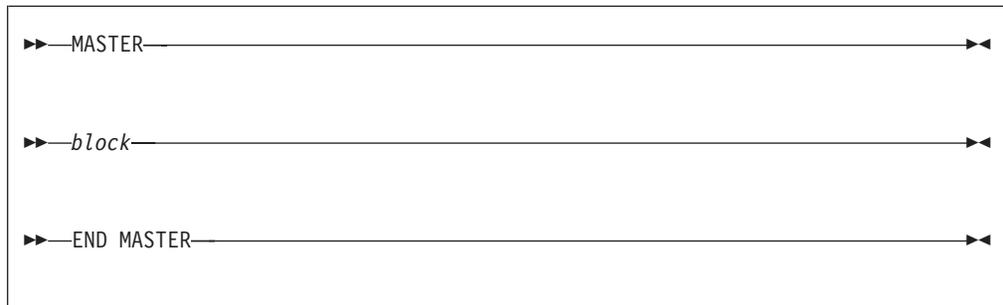
Purpose

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 **-qsmp** compiler option.

Syntax



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
!$OMP END MASTER
!$OMP BARRIER
!$OMP DO PRIVATE(J)
    DO J = 1, 10000

```

```

                A(J) = X + SIN(J*2.5)
            END DO
!$OMP END DO
!$OMP END PARALLEL
        END

```

Related Information

- *Loop parallelization* in the *XL Fortran Language Reference*
- **-qdirective** option in the *XL Fortran Compiler Reference*
- **-qsmp** option in the *XL Fortran Compiler Reference*
- “PARALLEL DO / END PARALLEL DO” on page 84
- “PARALLEL SECTIONS / END PARALLEL SECTIONS ” on page 87

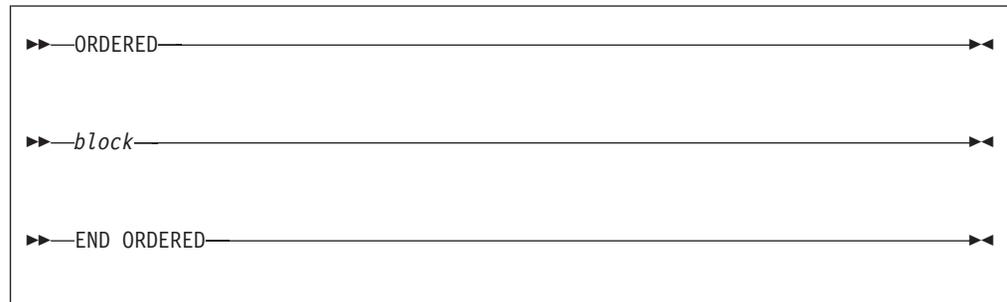
ORDERED / END ORDERED

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



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.

```
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.

```
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.

```
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
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 91 for more information.

Related Information

- *Loop parallelization* in the *XL Fortran Language Reference*
- **-qsmp** option in the *XL Fortran Compiler Reference*
- “**PARALLEL DO / END PARALLEL DO**” on page 84
- “**DO / END DO**” on page 71
- “**CRITICAL / END CRITICAL**” on page 69
- “**SCHEDULE** ” on page 91

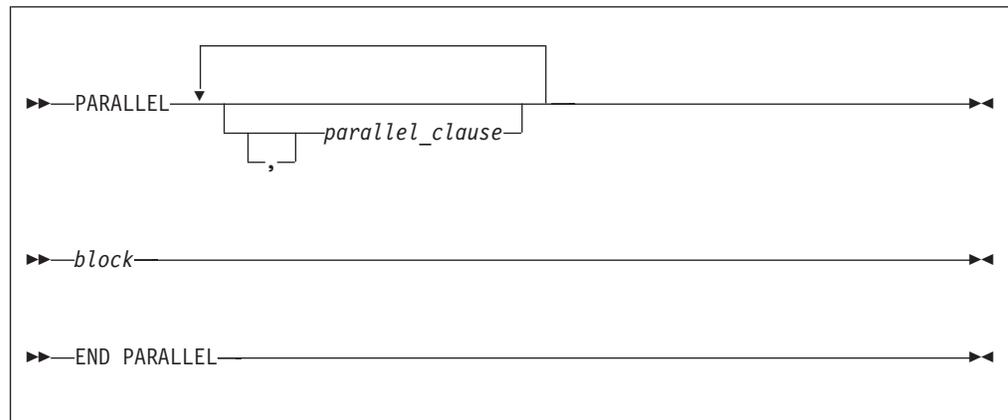
PARALLEL / END PARALLEL

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(—scalar_logical_expr—)
num_threads_clause
private_clause
reduction_clause
shared_clause

copyin_clause

See — “COPYIN” on page 112

default_clause

See — “DEFAULT” on page 114

if_clause

See — “IF” on page 116

firstprivate_clause

See — “FIRSTPRIVATE” on page 117.

num_threads_clause

See — “NUM_THREADS” on page 119.

private_clause

See — “PRIVATE” on page 121.

reduction_clause

See — “REDUCTION” on page 122

shared_clause

See — “SHARED” on page 127

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 225 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
!$OMP END PARALLEL
  END DO
!$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)
  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 Information

- “FLUSH” on page 75
- “PARALLEL DO / END PARALLEL DO” on page 84
- **INDEPENDENT** in the *XL Fortran Language Reference*
- “THREADPRIVATE” on page 104

- “DO / END DO” on page 71
- `-qdirective` option in the *XL Fortran Compiler Reference*
- `-qsmp` option in the *XL Fortran 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:

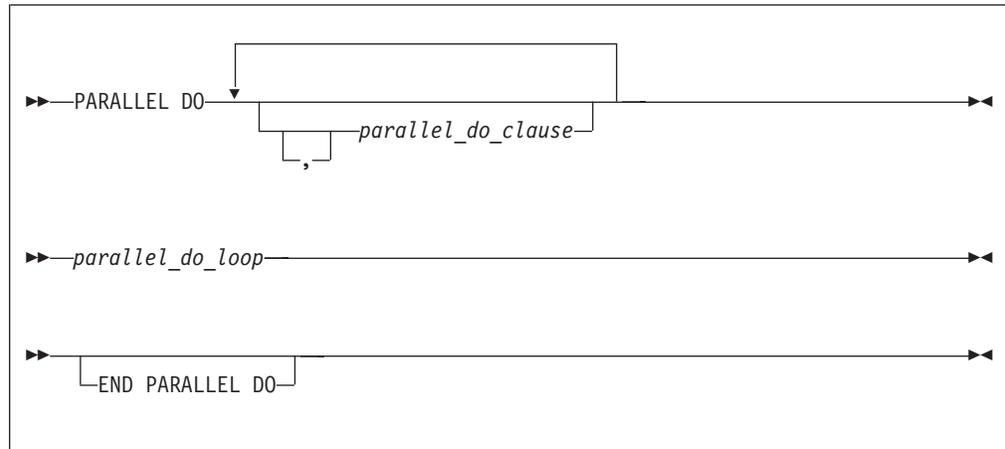
```
!$OMP PARALLEL
!$OMP DO
...
!$OMP ENDDO
!$OMP END PARALLEL
```

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.

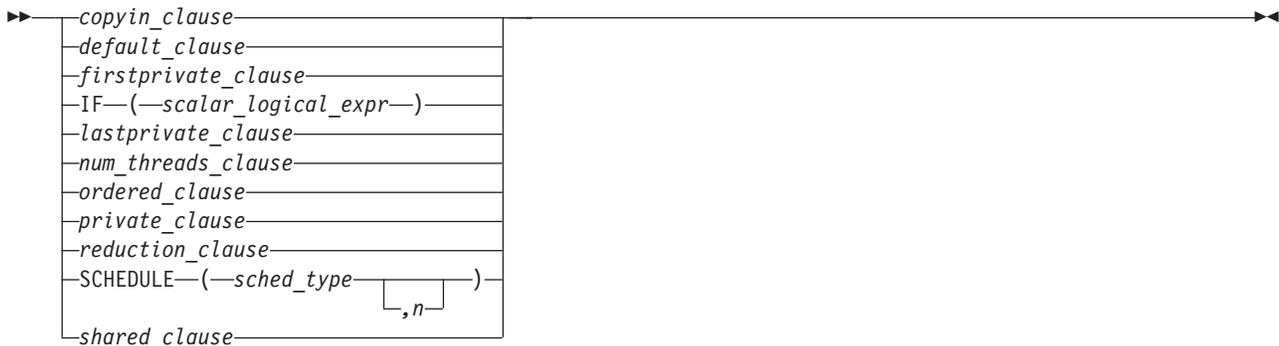
Type

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

Syntax



where *parallel_do_clause* is:



copyin_clause
See — “COPYIN” on page 112

default_clause
See — “DEFAULT” on page 114

if_clause
See — “IF” on page 116.

firstprivate_clause
See — “FIRSTPRIVATE” on page 117.

lastprivate_clause
See — “LASTPRIVATE” on page 118.

num_threads_clause
See — “NUM_THREADS” on page 119.

ordered_clause
See — “ORDERED” on page 120

private_clause
See — “PRIVATE” on page 121

reduction_clause
See — “REDUCTION” on page 122

schedule_clause
See — “SCHEDULE” on page 125

shared_clause
See — “SHARED” on page 127

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.

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=nested_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.

```
!$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.

```
!$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.

```
!$OMP PARALLEL DO SHARED (X)
  DO I = 1, 10
    A(I) = A(I) * I
```

```

!$OMP      CRITICAL
           X = X + A(I)
!$OMP      END CRITICAL
END DO

```

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

```

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

```

Related Information

- *Loop parallelization* in the *XL Fortran Language Reference*
- **-qdirective** option in the *XL Fortran Compiler Reference*
- **-qsmp** option in the *XL Fortran Compiler Reference*
- **DO** in the *XL Fortran Language Reference*
- “DO / END DO” on page 71
- **INDEPENDENT** in the *XL Fortran Language Reference*
- “ORDERED / END ORDERED” on page 79
- “PARALLEL / END PARALLEL ” on page 81
- “PARALLEL SECTIONS / END PARALLEL SECTIONS ”
- “SCHEDULE ” on page 91
- “THREADPRIVATE” on page 104

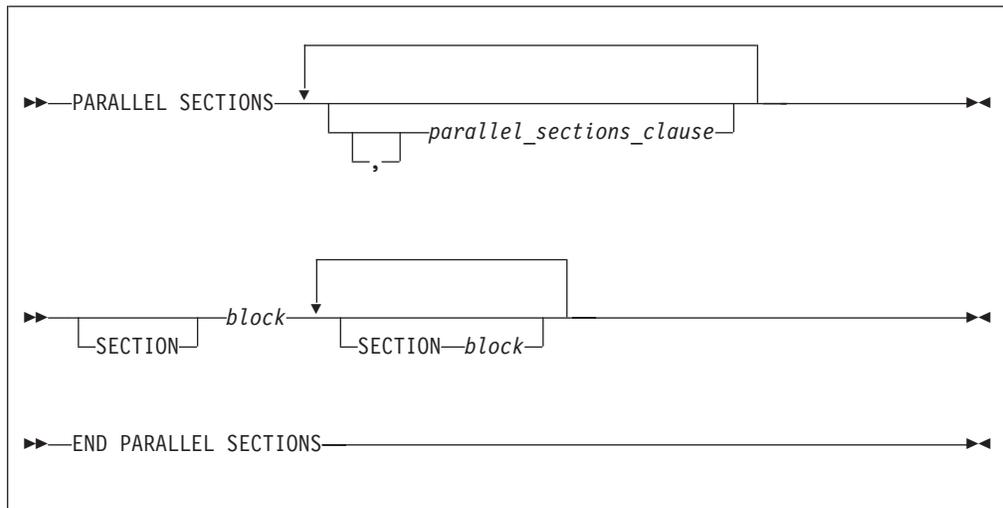
PARALLEL SECTIONS / END PARALLEL SECTIONS

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



where *parallel_sections_clause* is:



copyin_clause

See — “COPYIN” on page 112

default_clause

See — “DEFAULT” on page 114

firstprivate_clause

See — “FIRSTPRIVATE” on page 117.

if_clause

See — “IF” on page 116

lastprivate_clause

See — “LASTPRIVATE” on page 118.

num_threads_clause

See — “NUM_THREADS” on page 119.

private_clause

See — “PRIVATE” on page 121.

reduction_clause

See — “REDUCTION” on page 122

shared_clause

See — “SHARED” on page 127

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:

```
!$OMP PARALLEL SECTIONS
!$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.

```
!$OMP PARALLEL SECTIONS PRIVATE(I)
    DO I = 1, 100
        A(I) = A(I) * I
    END DO
!$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
    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 Information

- “PARALLEL / END PARALLEL ” on page 81
- “PARALLEL DO / END PARALLEL DO” on page 84
- **INDEPENDENT** in the *XL Fortran Language Reference*
- “THREADPRIVATE” on page 104
- **-qdirective** option in the *XL Fortran Compiler Reference*
- **-qsmp** option in the *XL Fortran 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.

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 56
5. runtime default (that is, **STATIC**)

Examples

Example 1. Given the following information:

```
number of iterations = 1000  
number of threads = 4
```

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:

```
chunk 1 = iterations 1 to 250  
chunk 2 = iterations 251 to 438  
chunk 3 = iterations 439 to 579  
chunk 4 = iterations 580 to 685  
chunk 5 = iterations 686 to 764  
chunk 6 = iterations 765 to 823  
chunk 7 = iterations 824 to 868  
chunk 8 = iterations 869 to 901  
chunk 9 = iterations 902 to 926  
chunk 10 = iterations 927 to 945  
chunk 11 = iterations 946 to 959  
chunk 12 = iterations 960 to 970  
chunk 13 = iterations 971 to 978  
chunk 14 = iterations 979 to 984  
chunk 15 = iterations 985 to 988
```

```
chunk 16 = iterations 989 to 991
chunk 17 = iterations 992 to 994
chunk 18 = iterations 995 to 996
chunk 19 = iterations 997 to 997
chunk 20 = iterations 998 to 998
chunk 21 = iterations 999 to 999
chunk 22 = iterations 1000 to 1000
```

A possible scenario for the division of work could be:

```
thread 1 executes chunks 1 5 10 13 18 20
thread 2 executes chunks 2 7 9 14 16 22
thread 3 executes chunks 3 6 12 15 19
thread 4 executes chunks 4 8 11 17 21
```

Example 2. Given the following information:

```
number of iterations = 100
number of threads = 4
```

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

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

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:

```
thread 1 executes chunks 1 5 9
thread 2 executes chunks 2 8
thread 3 executes chunks 3 6 10
thread 4 executes chunks 4 7
```

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 Information

- *DO* in the *XL Fortran Language Reference*

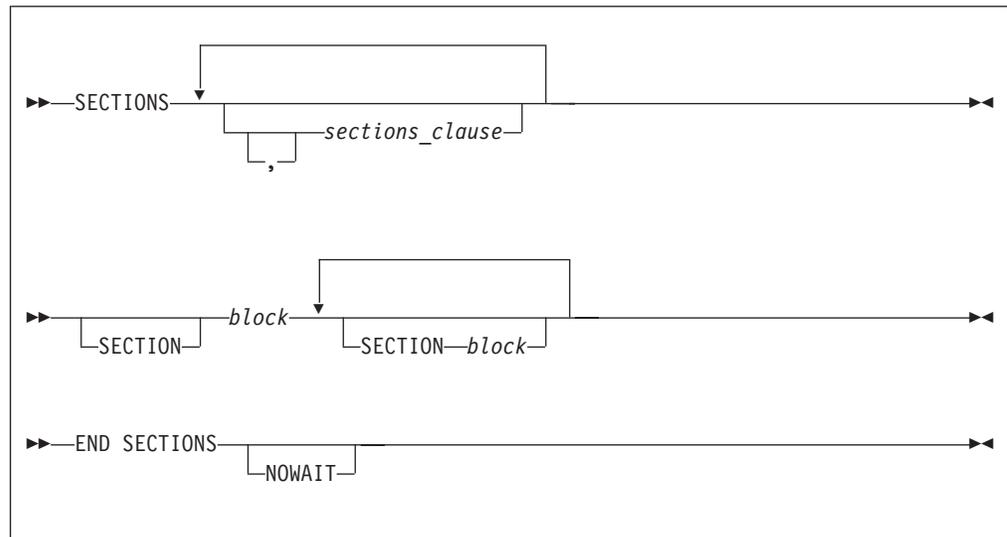
SECTIONS / END SECTIONS

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:



firstprivate_clause

See — “FIRSTPRIVATE” on page 117.

lastprivate_clause

See — “LASTPRIVATE” on page 118.

private_clause

See — “PRIVATE” on page 121.

reduction_clause

See — “REDUCTION” on page 122

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**, or **ORDERED** 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.

```

      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
! ...
!$OMP DO REDUCTION(-: SUM)
      DO J=I-1, 1, -1  ! The loop starts at 500 -- the last
                     ! value from the previous loop.
      SUM = SUM - B(J)
      END DO
!$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**.

```

!$OMP PARALLEL
!$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
!$OMP END SECTIONS
!$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 Information

- “PARALLEL / END PARALLEL ” on page 81
- “BARRIER” on page 68
- “PARALLEL DO / END PARALLEL DO” on page 84
- **INDEPENDENT** in the *XL Fortran Language Reference*
- “THREADPRIVATE” on page 104
- **-qdirective** option in the *XL Fortran Compiler Reference*
- **-qsmp** option in the *XL Fortran Compiler Reference*

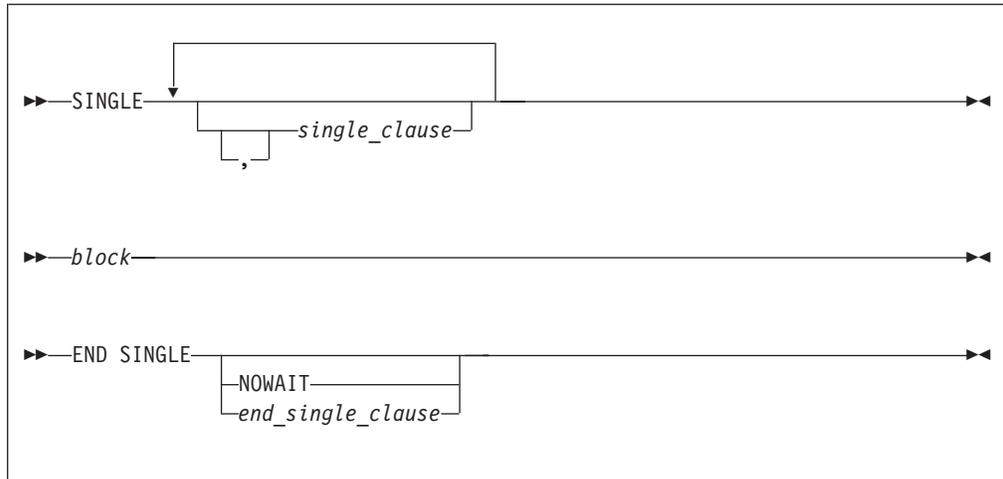
SINGLE / END SINGLE

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

See — “PRIVATE” on page 121.

firstprivate_clause

See — “FIRSTPRIVATE” on page 117.

where *end_single_clause* is:



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**, or **ORDERED** 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)
      DO I=1, 500
```

```

        CALL INITARR(B(I,:), I)      ! initialize the array B
    ENDDO
!$OMP END DO

!$OMP SINGLE                          ! employ only one thread
    DO I=1, 500
        SM = SM + SUM(B(J:J+1,I))
    ENDDO
!$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 Information

- “BARRIER” on page 68
- “CRITICAL / END CRITICAL” on page 69
- “FLUSH” on page 75
- “MASTER / END MASTER ” on page 78
- “PARALLEL / END PARALLEL ” on page 81

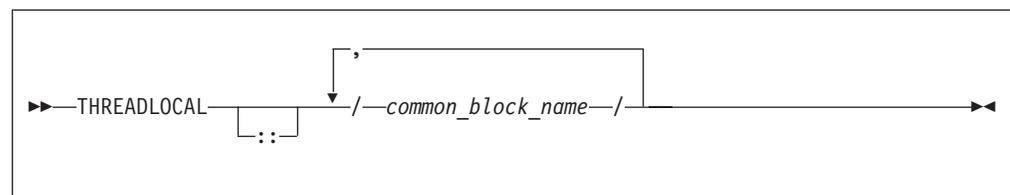
THREADLOCAL

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**, **xlf90_r**, **xlf95_r**, or **xlf2003_r** to link the necessary libraries.

Syntax



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 *pthread*s 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

```

! The SAVE attribute is required for the
! common block because the program requires
! that the block remain defined after
! library_routine1 is invoked.

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)
  are used to execute ANOTHER_SUB
  INTEGER AA
  COMMON /BLOCK/ J
  INTEGER :: J
  !IBM* THREADLOCAL /BLOCK/
  ...
  AA = J
  ! The value of 'J' is undefined.
END SUBROUTINE ANOTHER_SUB

```

! Multiple threads

! Each thread obtains a new copy of the
! common block /BLOCK/

Related Information

- **-qdirective** option in the *XL Fortran Compiler Reference*
- **-qinit** option in the *XL Fortran Compiler Reference*
- **COMMON** statement in the *XL Fortran Language Reference*
- Main program in the *XL Fortran Language Reference*

- One or more sample programs in the directory `/opt/ibmcmp/xlf/11.1/samples/modules/threadlocal` illustrate how to use the **THREADLOCAL** directive and create threads in C.

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

```
▶▶—THREADPRIVATE—(—threadprivate_entity_list—)▶▶
```

where *threadprivate_entity_list* is:

```
▶▶—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**.

```

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
!$OMP THREADPRIVATE(/BLK/) ! Array a is private to each thread.
! ...
  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
! ...
! Since each thread has its own copy of
! common block BLK, the assignment of
! array C has no effect on the copies of
! that block owned by other threads.
END SUBROUTINE SUB2

```

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.

```

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)
! The thread has its own copy of the
! common block ARR.

```

```

! ...
!$OMP SECTION                ! The thread has its own copy of the
! ...                        ! common block ARR.
        TEMP(1:100:2) = 1
        TEMP(2:100:2) = -2
        CALL SUBS(ARR2)
! ...
!$OMP END PARALLEL SECTIONS
! ...
        PRINT *, SUM(ARR1), SUM(ARR2)
END PROGRAM ABC

SUBROUTINE SUBS(X)
  INTEGER :: K, X(:), TEMP(100)
  COMMON /ARR/ TEMP
!$OMP THREADPRIVATE(/ARR/)
! ...
        DO K = 1, UBOUND(X, 1)
          X(K) = TEMP(K) + TEMP(K + 1)  ! The thread is accessing its
                                        ! own copy of
                                        ! the common block.
        ENDDO
! ...
END SUBROUTINE SUBS

```

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**.

```

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

!$OMP PARALLEL PRIVATE(I)
  I = OMP_GET_THREAD_NUM()
  IF (I .EQ. 0) THEN
    PRINT *, 'A(2) = ', A(2)
  ELSE IF (I .EQ. 1) THEN
    PRINT *, 'A(1) = ', A(1)
  END IF
!$OMP END PARALLEL

```

```

        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 Information

- **COMMON** statement in the *XL Fortran Language Reference*
- “OMP_DYNAMIC environment variable” on page 61
- “omp_set_dynamic(enable_expr)” on page 138
- “PARALLEL / END PARALLEL ” on page 81
- “PARALLEL DO / END PARALLEL DO” on page 84
- “PARALLEL SECTIONS / END PARALLEL SECTIONS ” on page 87

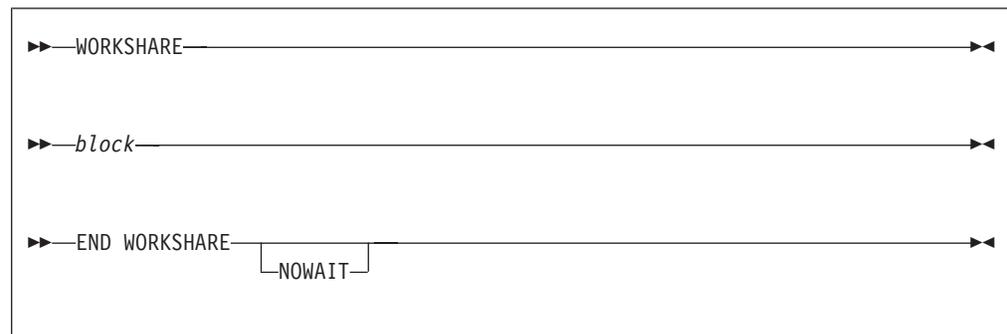
WORKSHARE

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



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 | • MATMUL | • PRODUCT |
| • ANY | • MAXLOC | • RESHAPE |
| • COUNT | • MAXVAL | • SPREAD |
| • CSHIFT | • MINLOC | • SUM |
| • DOT_PRODUCT | • MINVAL | • TRANSPOSE |
| • EOSHIFT | • PACK | • 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.

```
!$OMP WORKSHARE
  FORALL (I = 1 : N, AA(1, I) == 0) AA(1, I) = I
  BB = TRANSPOSE(AA)
  CC = MATMUL(AA, BB)
!$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.

```
!$OMP WORKSHARE
  WHERE (AA(1, :) /= 0.0) AA(1, :) = 1 / AA(1, :)
  DD = TRANS(AA(1, :))
!$OMP END WORKSHARE

ELEMENTAL REAL FUNCTION TRANS(ELM) RESULT(RES)
REAL, INTENT(IN) :: ELM
RES = ELM * ELM + 4
END FUNCTION
```

Related Information

- “ATOMIC” on page 66
- “BARRIER” on page 68

- “CRITICAL / END CRITICAL” on page 69
- “PARALLEL WORKSHARE / END PARALLEL WORKSHARE” on page 90
- `-qsmp` option in the *XL Fortran Compiler Reference*

OpenMP directive clauses

The following OpenMP directive clauses allow you to specify the scope attributes of variables within a parallel construct. The **IF**, **NUM_THREADS**, **ORDERED**, and **SCHEDULE** clauses, also in this section, allow you to control the parallel environment of a parallel region. See the detailed directive descriptions for more information.

COPYIN	FIRSTPRIVATE	PRIVATE
COPYPRIVATE	LASTPRIVATE	REDUCTION
DEFAULT	NUM_THREADS	SCHEDULE
IF	ORDERED	SHARED

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 procedure referenced within the dynamic extent of a parallel region have an implicit **SHARED** attribute, unless they are **THREADLOCAL** or **THREADPRIVATE** common blocks and module variables.

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.

IBM Extension

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**.

End of IBM Extension

To maintain full compliance with the OpenMP API Version 2.5 standard, ensure that a **POINTER** variable applies only to a **PRIVATE** clause.

COPYIN

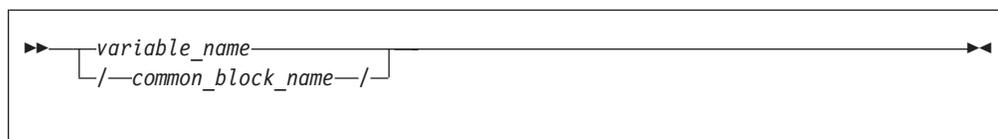
Purpose

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

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.

The **COPYIN** clause applies to:

- **PARALLEL**
- **PARALLEL DO**
- **PARALLEL SECTIONS**
- **PARALLEL WORKSHARE**

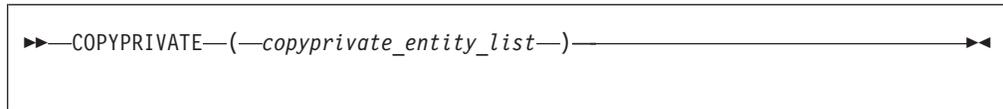
COPYPRIVATE

Purpose

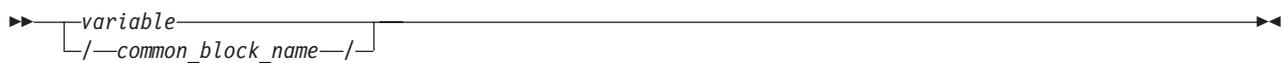
If you specify the **COPYPRIVATE** clause, the value of a private variable or pointer to a shared object from one thread in a team is copied into the corresponding

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_entity



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.

A **COPYPRIVATE** clause applies to the following directives:

- **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 **PRIVATE**, **SHARED**, or no default scope attribute.

Syntax

▶▶—**DEFAULT**—(—*default_scope_attr*—)————▶▶

default_scope_attr
is one of **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(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 **POINTEES** 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 **POINTEES** 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.

The **DEFAULT** clause applies to:

- **PARALLEL**

- PARALLEL DO
- PARALLEL SECTIONS
- PARALLEL WORKSHARE

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

```

IF

Purpose

If you specify the **IF** clause, the runtime environment performs a test to determine whether to run the block in serial or parallel. If *scalar_logical_expression* is true, then the block is run in parallel; if not, then the block is run in serial.

Syntax

▶—IF—(*scalar_logical_expression*)—▶

Rules

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

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.

The **IF** clause applies to the following directives:

- PARALLEL/END PARALLEL
- PARALLEL DO / END PARALLEL DO
- PARALLEL SECTIONS / END PARALLEL SECTIONS

- PARALLEL WORKSHARE / END PARALLEL

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 pointer
- 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.

The **FIRSTPRIVATE** clause applies to the following directives:

- **DO**

- PARALLEL
- PARALLEL DO
- PARALLEL SECTIONS
- PARALLEL WORKSHARE
- SECTIONS
- SINGLE

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.

The **LASTPRIVATE** clause applies to the following directives:

- **DO**
- **PARALLEL DO**
- **PARALLEL SECTIONS**
- **SECTIONS**

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
```

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

▶▶—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.

The **NUM_THREADS** clause applies to the following work-sharing constructs:

- **PARALLEL**
- **PARALLEL DO**
- **PARALLEL SECTIONS**
- **PARALLEL WORKSHARE**

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

▶▶—ORDERED————▶▶

Rules

The **ORDERED** clause applies to the following directives:

- “DO / END DO” on page 71
- “PARALLEL DO / END PARALLEL DO” on page 84

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.

The **PRIVATE** clause applies to the following directives:

- **DO**
- **PARALLEL**
- **PARALLEL DO**
- **PARALLEL SECTIONS**
- **SECTIONS**
- **SINGLE**
- **PARALLEL WORKSHARE**

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**.

```

      INTEGER :: ARR(10), J = 17
      ISTFNC() = J

!$OMP PARALLEL DO 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

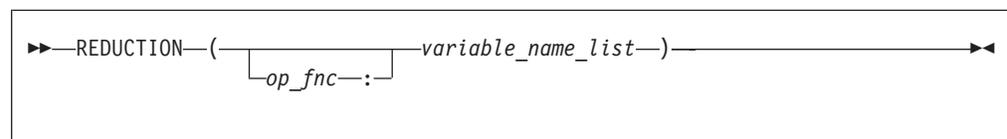
```

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.

Syntax



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:

▶▶—*reduction_var_ref*—=*expr*—*reduction_op*—*reduction_var_ref*—▶▶

▶▶—*reduction_var_ref*—=*reduction_var_ref*—*reduction_op*—*expr*—▶▶

▶▶—*reduction_var_ref* =—*reduction_function*—(*expr*,—*reduction_var_ref*)—▶▶

▶▶—*reduction_var_ref* =—*reduction_function*—(*reduction_var_ref*,—*expr*)—▶▶

where:

reduction_var_ref

is a variable or subobject of a variable that appears in a **REDUCTION** clause

reduction_op

is one of the intrinsic operators: +, −, *, **.AND.**, **.OR.**, **.EQV.**, **.NEQV.**, or **.XOR.**

reduction_function

is one of the intrinsic procedures: **MAX**, **MIN**, **IAND**, **IOR**, or **IEOR**.

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.

Intrinsic Operator	Initialization
+	0
*	1
-	0
.AND.	.TRUE.
.OR.	.FALSE.
.EQV.	.TRUE.
.NEQV.	.FALSE.
.XOR.	.FALSE.
Intrinsic Procedure	Initialization
MAX	Smallest representable number
MIN	Largest representable number
IAND	All bits on
IOR	0
IEOR	0

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_refs* that appear in a **REDUCTION** statement must be lexically identical.
- You cannot use the following form of the **REDUCTION** statement:
reduction_var_ref = *expr operator reduction_var_ref*

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* **\$OMP**.

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.

The OpenMP implementation of the **REDUCTION** clause applies to:

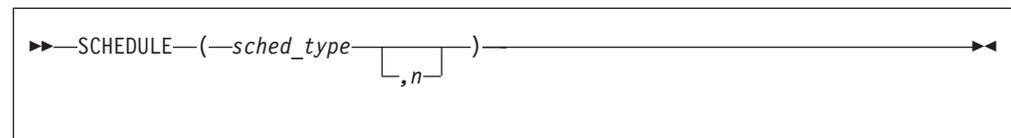
- **DO**
- **PARALLEL**
- **PARALLEL DO**
- **PARALLEL SECTIONS**
- **PARALLEL WORKSHARE**
- **SECTIONS**

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



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* **\$OMP**, do not specify the scheduling type **AFFINITY**.

AFFINITY

The iterations of a loop are initially divided into *number_of_threads* partitions, containing

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

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 $\text{CEILING}(\text{number_of_iterations_remaining_in_partition} / 2)$

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:

`CEILING(number_of_iterations / number_of_threads)`

iterations. Subsequent chunks are proportional to:

`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 **DO** construct. Any function reference in the n expression must not have side effects.

The value of the n parameter on the **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 **XLSPMPOPTS**. If no scheduling type is specified using that variable, then the default scheduling type used is **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

CEILING(number_of_iterations / 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.

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 **SCHEDULE** clause more than once for a particular **DO** directive.

The **SCHEDULE** clause applies to the following directives:

- “DO / END DO” on page 71
- “PARALLEL DO / END PARALLEL DO” on page 84

SHARED

Purpose

All sections use the same copy of the variables and common blocks you specify in *data_scope_entity_list*.

The **SHARED** clause specifies variables that must be available to all threads. If you specify a variable as **SHARED**, you are stating that all threads can safely share a single copy of the variable.

Syntax

```
▶▶—SHARED—(—data_scope_entity_list—)————▶▶
```

data_scope_entity

```
▶▶—named_variable————▶▶  
└—/common_block_name/—┘
```

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**.

The **SHARED** clause applies to:

- **PARALLEL**
- **PARALLEL DO**
- **PARALLEL SECTIONS**
- **PARALLEL WORKSHARE**

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.

```

      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
CONTAINS
  SUBROUTINE SUB1(ARR)
    INTEGER:: ARR(1:4)
    DO I=1, 4
      ARR(I) = I
    END DO
  END SUBROUTINE
END
```

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 13. OpenMP execution environment routines

omp_get_dynamic	omp_get_thread_num
omp_get_max_threads	omp_in_parallel
omp_get_nested	omp_set_dynamic
omp_get_num_procs	omp_set_nested
omp_get_num_threads	omp_set_num_threads

Included in the OpenMP runtime library are two routines that support a portable wall-clock timer.

Table 14. OpenMP timing routines

omp_get_wtick	omp_get_wtime
----------------------	----------------------

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 15. OpenMP simple lock routines

omp_destroy_lock	omp_test_lock
omp_init_lock	omp_unset_lock
omp_set_lock	

Table 16. OpenMP nestable lock routines

<code>omp_destroy_nest_lock</code>	<code>omp_test_nest_lock</code>
<code>omp_init_nest_lock</code>	<code>omp_unset_nest_lock</code>
<code>omp_set_nest_lock</code>	

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.

```

      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 uninitialized 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

`.TRUE.` if dynamic thread adjustment by the runtime environment is enabled.
`.FALSE.` otherwise.

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

```
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.

```
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)

!$OMP MASTER
  NP = omp_get_thread_num()
  CALL WORK('in master', NP)
!$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

```
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 139 and the OMP_NESTED environment variable for more information.)

Class

Function.

Argument Type and Attributes

None.

Result Type and Attributes

Default logical.

Result Value

.TRUE. if called from the dynamic extent of a region executing in parallel. **.FALSE.** 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.

```

        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 you call `omp_set_dynamic` with a *scalar_logical_expression* that evaluates 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 you call the subroutine with a *scalar_logical_expression* which evaluates 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.

```
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 you call the subroutine with a *scalar_logical_expression* that evaluates 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 you call the subroutine with a *scalar_logical_expression* that evaluates 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.

`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

```
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 *scalar_integer_expression* that you pass to the subroutine 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 138), **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
scalar_integer_expression

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 variable 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. 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 logical.

Result Value

.TRUE. if the function was able to set the lock. **.FALSE.** otherwise.

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

```
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
!$OMP END DO
!$OMP END PARALLEL
CALL omp_destroy_lock (LCK_X)
END
```

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

```

End of IBM Extension

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 Linux pthreads library. The Linux 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 Linux pthreads library routine name or type definition name.

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 Linux pthreads library. However, some of the pthread routines have no corresponding procedures in this module because they are not supported on Linux. 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 Linux error code names. For example, **EINVAL** is the Fortran constant name of the error code **EINVAL** on Linux. For a complete list of these error codes, refer to the file **/usr/include/errno.h** on Linux.

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_getstack(attr, stackaddr, 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_setstack(attr, stackaddr, 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_getpshared(mattr, pshared)
- f_pthread_mutexattr_gettype(mattr, type)
- f_pthread_mutexattr_init(mattr)

- `f_pthread_mutexattr_setpshared(mattr, pshared)`
- `f_pthread_mutexattr_settype(mattr, type)`

Functions that perform operations on mutex objects

- `f_pthread_mutex_destroy(mutex)`
- `f_pthread_mutex_init(mutex, mattr)`
- `f_pthread_mutex_lock(mutex)`
- `f_pthread_mutex_trylock(mutex)`
- `f_pthread_mutex_unlock(mutex)`

Functions that perform operations on attribute objects of condition variables

- `f_pthread_condattr_destroy(cattr)`
- `f_pthread_condattr_getpshared(cattr, pshared)`
- `f_pthread_condattr_init(cattr)`
- `f_pthread_condattr_setpshared(cattr, pshared)`

Functions that perform operations on condition variable objects

- `f_maketime(delay)`
- `f_pthread_cond_broadcast(cond)`
- `f_pthread_cond_destroy(cond)`
- `f_pthread_cond_init(cond, cattr)`
- `f_pthread_cond_signal(cond)`
- `f_pthread_cond_timedwait(cond, mutex, timeout)`
- `f_pthread_cond_wait(cond, mutex)`

Functions that perform operations on thread-specific data

- `f_pthread_getspecific(key, arg)`
- `f_pthread_key_create(key, dtr)`
- `f_pthread_key_delete(key)`
- `f_pthread_setspecific(key, arg)`

Functions and subroutines that perform operations to control thread cancelability

- `f_pthread_setcancelstate(state, oldstate)`
- `f_pthread_setcanceltype(type, oldtype)`
- `f_pthread_testcancel()`

Functions that perform operations on read-write lock attribute objects

- `f_pthread_rwlockattr_destroy(rwattr)`
- `f_pthread_rwlockattr_getpshared(rwattr, pshared)`
- `f_pthread_rwlockattr_init(rwattr)`
- `f_pthread_rwlockattr_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_gettime(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

`delay` INTEGER(4), INTENT(IN)

Result Type and Attributes

TYPE (`f_timespec`)

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 recollectd by the system.

Class

Function

Argument Type and Attributes

`attr` TYPE(`f_pthread_attr_t`), 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_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_thread_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. This is the system default.

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.

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_thread_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

attr TYPE(f_thread_attr_t), INTENT(IN)

guardsize

INTEGER(KIND=register_size), INTENT(IN)

where *register_size* is the size of a pointer, in bytes, in the current addressing mode. That 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_thread_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

attr TYPE(f_thread_attr_t), INTENT(OUT)

inherit

INTEGER(4)

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.

f_thread_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**.

Class

Function

Argument Type and Attributes

attr TYPE(f_thread_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 the following error.

EINVAL The argument **attr** is invalid.

f_thread_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**.

Class

Function

Argument Type and Attributes

attr TYPE(f_thread_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 the following error.

EINVAL The argument **attr** is invalid.

f_thread_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_thread_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.

scope Contains the following value:

PTHREAD_SCOPE_SYSTEM:

the thread will compete for system resources on a system wide scope.

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_thread_attr_getstack(attr, stackaddr, ssize)

Purpose

Retrieves the values of the **stackaddr** and **ssize** arguments from the thread attribute object **attr**.

Class

Function

Argument Type and Attributes

attr TYPE(f_thread_attr_t), INTENT(IN)

stackaddr
Integer pointer, INTENT(OUT)

ssize INTEGER(KIND=register_size), INTENT(OUT)

where *register_size* is the size of a pointer, in bytes, in the current addressing mode. That 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
One or more of the supplied arguments is invalid.

f_thread_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.

Class

Function

Argument Type and Attributes

attr TYPE(f_thread_attr_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 **attr** is invalid.

f_pthread_attr_setdetachstate(attr, detach)

Purpose

This function can be used to set the detach state attribute in the thread attribute object **attr**.

Class

Function

Argument Type and Attributes

attr TYPE(f_pthread_attr_t), INTENT(OUT)

detach INTEGER(4), INTENT(IN)

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_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.

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 **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**.

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 the size of a pointer, in bytes, in the current addressing mode. That 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_thread_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_thread_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 the following error.

EINVAL

The argument **inherit** is invalid.

f_thread_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_thread_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 the following error.

EINVAL

The argument **param** is invalid.

f_thread_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 error.

EINVAL

The argument **policy** is invalid.

f_thread_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_thread_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 the following error.

EINVAL

The argument **scope** is invalid.

f_thread_attr_setstack(attr, stackaddr, ssize)

Purpose

Use this function to set the stack address and stack size attributes in the pthread attribute object **attr**. The **stackaddr** argument represents the stack address as an Integer pointer. The **stacksize** argument is an integer that represents the size of the stack in bytes. When creating a thread using the attribute object **attr**, the system allocates a minimum stack size of **stacksize** bytes.

Class

Function

Argument Type and Attributes

attr TYPE(f_thread_attr_t), INTENT(INOUT)

stackaddr

Integer pointer, INTENT(IN)

ssize INTEGER(KIND=register_size)

where *register_size* is the size of a pointer, in bytes, in the current addressing mode. That 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 value of one or both of the supplied arguments is invalid.

EACCES

The stack pages specified are not readable by the thread.

f_pthread_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 cancellation 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 cancellation request will be put on hold until the target thread reaches its next cancellation 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.

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.

ESRCH

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 **arg** is a generic argument that can be of any type and any rank. 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.

If the actual argument **arg** is an array section, the corresponding dummy argument in subroutine **cleanup** must be an assumed-shape array. Otherwise, the result is unpredictable.

If the actual argument **arg** has the pointer attribute that points to an array or array section, the corresponding dummy argument in subroutine **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 **f_pthread_cleanup_pop**.

The argument **flag** must be used to convey the property of argument **arg** exactly to the system.

Class

Function

Argument Type and Attributes

cleanup

A subroutine that has one dummy argument.

flag Flag is an INTEGER(4), INTENT(IN) argument that can contain one of, or a combination of, the following constants:

FLAG_CHARACTER:

if the entry subroutine **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 **FLAG_DEFAULT** value should be included instead.

FLAG_ASSUMED_SHAPE:

if the entry subroutine **cleanup** has a dummy argument that is an assumed-shape array of any rank, this flag value must be included to indicate this fact.

FLAG_DEFAULT:

otherwise, this flag value is needed.

arg A generic argument that can be of any type, kind, and rank.

Result Type and Attributes

INTEGER(4)

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_thread_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_thread_cond_wait** will be blocked, and will wait on the condition variable **cond**.

Class

Function

Argument Type and Attributes

cond TYPE(f_thread_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_thread_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.

Class

Function

Argument Type and Attributes

cond TYPE(f_thread_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.

EBUSY The condition variable **cond** is being used by another thread.

f_thread_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.

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_thread_cond_t), INTENT(INOUT)

cattr TYPE(f_thread_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_thread_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_thread_cond_wait** will be blocked, and will wait on the condition variable **cond**.

Class

Function

Argument Type and Attributes

cond TYPE(f_thread_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_thread_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_thread_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**.

Class

Function

Argument Type and Attributes

cond TYPE(f_thread_cond_t), INTENT(INOUT)

mutex TYPE(f_thread_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.

ETIMEDOUT

The waiting deadline was reached before the condition occurred.

f_thread_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_thread_cond_t), INTENT(INOUT)

mutex TYPE(f_thread_mutex_t), INTENT(INOUT)

Result Type and Attributes

INTEGER(4)

Result Value

This function returns 0.

f_thread_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

cattr TYPE(f_thread_condattr_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 **cattr** is invalid.

EBUSY

Returns **EBUSY** if threads are waiting on the for the condition to occur.

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

cattr TYPE(f_pthread_condattr_t), INTENT(IN)

pshared

INTEGER(4), INTENT(OUT)

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.

f_pthread_condattr_init(cattr)

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.

Class

Function

Argument Type and Attributes

cattr TYPE(f_pthread_condattr_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 condition variable attributes object.

f_thread_condattr_setshared(cattr, pshared)

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**.

Class

Function

Argument Type and Attributes

cattr TYPE(f_thread_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_thread_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_thread_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 **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 **arg** is a generic argument that can be of any type and any rank. 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.

If the actual argument **arg** is an array section, the corresponding dummy argument in subroutine **ent** must be an assumed-shape array. Otherwise, the result is unpredictable.

If the actual argument **arg** has the pointer attribute that points to an array or array section, the corresponding dummy argument in subroutine **ent** must have a pointer attribute or be an assumed-shape array. Otherwise, the result is unpredictable.

Class

Function

Argument Type and Attributes

thread TYPE(f_thread_t), INTENT(OUT)

On successful completion of the function, **f_thread_create** stores the ID of the created thread in the **thread**.

attr TYPE(f_thread_attr_t), INTENT(IN)

flag INTEGER(4), INTENT(IN)

The argument **flag** must convey the property of the argument **arg** exactly to the system. The argument **flag** can be one of, or a combination of, the following constants:

FLAG_CHARACTER:

if the entry subroutine **ent** 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 **FLAG_DEFAULT** value should be included instead.

FLAG_ASSUMED_SHAPE:

if the entry subroutine **ent** has a dummy argument which is an assumed-shape array of any rank, this flag value must be included to indicate this fact.

FLAG_DEFAULT:

otherwise, this flag value is needed.

ent A subroutine that has one dummy argument.

arg A generic argument that can be of any type, kind, and rank.

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

thread TYPE(f_pthread_t), INTENT(IN)

Result Type and Attributes

INTEGER(4)

Result Value

On successful completion, this function returns 0. Otherwise, this function returns the following error.

ESRCH

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

thread1

TYPE(f_thread_t), INTENT(IN)

thread2

TYPE(f_thread_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_thread_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_thread_exit** will not automatically free all of the memory allocated to a thread. To avoid memory leaks, finalization must be handled separately from **f_thread_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_thread_getconcurrency()

Purpose

This function returns the value of the concurrency level set by a previous call to the **f_thread_setconcurrency** function. If the **f_thread_setconcurrency** function was not previously called, this function returns zero to indicate that the system is maintaining the concurrency level.

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.

`f_pthread_getschedparam(thread, policy, 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 `thread`. Its scheduling policy will be returned through argument `policy` and its scheduling property through argument `param`. The `sched_priority` field in `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

`thread` TYPE(`f_pthread_t`), INTENT(IN)

`policy` INTEGER(4), INTENT(OUT)

`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.

ESRCH

The target thread is invalid or has already terminated.

EFAULT

The `policy` or `param` points are outside the process memory space

`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_thread_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_thread_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_thread_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_thread_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.

ENOMEM 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.

Class

Function

Argument Type and Attributes

thread TYPE(f_pthread_t), INTENT(INOUT)

sig 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.

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

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.
EINVAL	The argument mutex is invalid.

f_thread_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_thread_mutex_t), INTENT(OUT)

mattr TYPE(f_thread_mutexattr_t), INTENT(IN), OPTIONAL

Result Type and Attributes

INTEGER(4)

Result Value

This function always returns 0.

f_thread_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_thread_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_thread_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_thread_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_thread_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.

EINVAL The argument **mutex** is invalid.

f_thread_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_thread_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_thread_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

mattr TYPE(f_thread_mutexattr_t), INTENT(INOUT)

Result Type and Attributes

INTEGER(4)

Result Value

This function always returns 0.

f_thread_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

mattr TYPE(f_pthread_mutexattr_t), INTENT(IN)

type INTEGER(4), INTENT(OUT)

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_thread_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_thread_mutexattr_t), INTENT(OUT)

Result Type and Attributes

INTEGER(4)

Result Value

This function returns 0.

f_thread_mutexattr_setshared(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_thread_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.

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_thread_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_thread_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_thread_once_t`), INTENT(INOUT)

`initr` A subroutine that has no dummy arguments.

Result Type and Attributes

INTEGER(4)

Result Value

This function returns 0.

f_thread_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_thread_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_thread_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.

f_thread_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.

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

rwlock

TYPE(f_thread_rwlock_t), INTENT(OUT)

rwattr TYPE(f_thread_rwlockattr_t), INTENT(IN), OPTIONAL

Result Type and Attributes

INTEGER(4)

Result Value

This function returns 0.

f_thread_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_thread_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_thread_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).

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.

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`.

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

`rwlock`

TYPE(`f_pthread_rwlock_t`), INTENT(INOUT)

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, the following error 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.

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`

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, the following error will be returned:

EBUSY

The read-write lock could not be acquired for writing because it is already locked for reading or writing.

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.

EPERM

The current thread does not own the read-write lock.

f_thread_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_thread_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_thread_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_thread_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_thread_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_thread_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**. **pshared** will contain one of the following values:

Class

Function

Argument Type and Attributes

rwattr TYPE(f_thread_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_thread_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_thread_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_thread_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**.

Class

Function

Argument Type and Attributes

rwattr TYPE(f_thread_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 one of the following errors.

EINVAL

The argument **rwattr** is invalid.

ENOSYS

The value of **pshared** is equal to **pthread_process_shared**.

f_thread_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.

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:

cancelation request will be delayed until a cancelation point.

PTHREAD_CANCEL_ASYNCHRONOUS:

cancelation 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:

cancelation request will be delayed until a cancelation point.

PTHREAD_CANCEL_ASYNCHRONOUS:

cancelation 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.

Class

Function

Argument Type and Attributes

new_level

INTEGER(4), INTENT(IN)

Result Type and Attributes

INTEGER(4)

Result Value

This function returns 0.

f_thread_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 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.

Class

Function

Argument Type and Attributes

thread TYPE(f_thread_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

ENOSYS	The POSIX priority scheduling option is not implemented on Linux.
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_thread_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_thread_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_thread_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_thread_testcancel()

Purpose

This subroutine provides a cancellation point in a thread. When it is called, any pending cancellation 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 Linux system data structure **sched_param**, which is a system data type.

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 Linux 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

```

Class

Data Type

End of IBM Extension

Chapter 8. Interlanguage calls

This section provides details on performing interlanguage calls from your Fortran application, allowing you to call routines that were 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:

- Both the underscore (`_`) and the dollar sign (`$`) are valid characters anywhere in names.

Because names that begin with an underscore are reserved for the names of library routines, do not use an underscore as the first character of a Fortran external name.

To avoid conflicts between Fortran and non-Fortran function names, you can compile the Fortran program with the `-qextname` option. This option adds an underscore to the end of the Fortran names. Then use an underscore as the last character of any non-Fortran procedures that you want to call from Fortran.

- Names can be up to 250 characters long.
- Program and symbolic names are interpreted as all lowercase by default. If you are writing new non-Fortran code, use all-lowercase procedure names to simplify calling the procedures from Fortran.

You can use the `-U` option or the `@PROCESS MIXED` directive if you want the names to use both uppercase and lowercase:

```
@process mixed
  external C_Func      ! With MIXED, we can call C_Func, not just c_func.
  integer aBC, ABC    ! With MIXED, these are different variables.
  common /xYz/ aBc    ! The same applies to the common block names.
  common /XYZ/ ABC    ! xYz and XYZ are external names that are
                    ! visible during linking.
end
```

- Names for module procedures are formed by concatenating `__` (two underscores), the module name, `_IMOD_` (for intrinsic modules) or `_NMOD_` (for non-intrinsic modules), and the name of the module procedure. For example, module procedure `MYPROC` in module `MYMOD` has the external name `__mymod_NMOD_myproc`.

Note: Symbolic debuggers and other tools should account for this naming scheme when debugging XL Fortran programs that contain module procedures. For example, some debuggers default to lowercase for program and symbolic names. This behavior should be changed to use mixed case when debugging XL Fortran programs with module procedures.

- The XL compilers generate code that uses `main` as an external entry point name. You can only use `main` as an external name in these contexts:
 - A Fortran program or local-variable name. (This restriction means that you cannot use `main` 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 *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.

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.

Related information: For more information on the `flush_` and `getfd` procedures, see the *Service and utility procedures* section in the *XL Fortran Language Reference*. For more information on the `buffering` runtime option, see *Setting runtime options* in the *XL Fortran Compiler Reference*.

Mixing Fortran and C++

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 `-I` options for the XL Fortran library directories and libraries.

```
program main

integer idim, idim1

idim = 35
idim1 = 45

write(6,*) 'Inside Fortran calling first C function'
call cfun(idim)
write(6,*) 'Inside Fortran calling second C function'
call cfun1(idim1)
write(6,*) 'Exiting the Fortran program'
end
```

Figure 7. Main Fortran program that calls C++ (main1.f)

```
#include <stdio.h>
#include "cplus.h"

extern "C" void cfun(int *idim){
    printf("%sInside C function before creating C++ Object\n");
    int i = *idim;
    junk<int>* jj= new junk<int>(10,30);
    jj->store(idim);
    jj->print();
    printf("%sInside C function after creating C++ Object\n");
    delete jj;
    return;
}

void cfun1(int *idim1) {
    printf("%sInside C function cfun1 before creating C++ Object\n");
    int i = *idim1;
    temp<double> *tmp = new temp<double>(40, 50.54);
    tmp->print();
    printf("%sInside 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 << endl; }};

template<class T> class temp {

private:
    int internal;
    T  temp_var;

public:
    temp(int i, T j): internal(i),temp_var(j)
        {cout <<"***Inside C++ temp Constructor" <<endl;}

    ~temp()      {cout <<"***Inside C++ temp destructor" <<endl;}

    void print(void) {cout << internal << "\t" << temp_var << endl;}};

```

Figure 9. C++ code called from Fortran (*cplus.h*)

Compiling this program, linking it with the `xLC` or `g++` command, and running it produces this output:

```

Inside Fortran calling first C function
%Inside C function before creating C++ Object
***Inside C++ constructor
10    30    35
%Inside C function after creating C++ Object
***Inside C++ Destructor
Inside Fortran calling second C function
%Inside C function cfun1 before creating C++ Object
***Inside C++ temp Constructor
40    50.54
%Inside C function after creating C++ temp object
***Inside C++ temp destructor
Exiting the Fortran program

```

Making calls to C functions work

When you pass an argument to a subprogram call, the usual Fortran convention is to pass the address of the argument. Many C functions expect arguments to be passed as values, however, not as addresses. For these arguments, specify them as `%VAL(argument)` in the call to C, 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 197 and %VAL and %REF in the *XL Fortran Language Reference* for more details.

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 sections 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

Table 17. Corresponding data types in Fortran and C. When calling Fortran procedures, the C routines must pass arguments as pointers to the types listed in this table.

XL Fortran Data Types	XL C/C++ Data Types
INTEGER(1), BYTE	signed char
INTEGER(2)	signed short
INTEGER(4)	signed int
INTEGER(8)	signed long long
REAL, REAL(4)	float
REAL(8), DOUBLE PRECISION	double
REAL(16)	long double (see note 1)
COMPLEX, COMPLEX(4)	float _Complex
COMPLEX(8), DOUBLE COMPLEX	double _Complex
COMPLEX(16)	long double _Complex (see note 1)
LOGICAL(1)	unsigned char
LOGICAL(2)	unsigned short
LOGICAL(4)	unsigned int
LOGICAL(8)	unsigned long long
CHARACTER	char
CHARACTER(n)	char[n]
Integer POINTER	void *
Array	array
Sequence-derived type	structure (with C/C++ -qalign=packed option)
Notes: 1. Requires C/C++ compiler -qlongdbl option.	

Notes:

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 197.
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.

Related information: One or more sample programs under the directory `/opt/ibmcmp/xlf/11.1/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 to get packed structures.

```
program cstruct                                struct mystuff {
real(8) a,d                                    double a;
integer b,c                                    int b,c;
.                                                double d;
.                                                };
common /mystuff/ a,b,c,d
.
.
end                                             main() {
.                                                }
.                                                }
```

If you do not have a specific need for a named common block, you can create a sequence-derived type with the same one-to-one mapping as a C structure and pass it as an argument to a C function. You must compile the C source file to get packed structures or put `#pragmas` 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 "Sample Fortran programs," on page 249.

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.

If you are writing both parts of the mixed-language program, you can make the C routines deal with the extra Fortran length argument, or you can suppress this extra argument by passing the string using the **%REF** function. If you use **%REF**, which you typically would for pre-existing C routines, you need to indicate where the string ends by concatenating a null character to the end of each character string that is passed to a C routine:

```
! Initialize a character string to pass to C.
  character*6 message1 /'Hello\0'/
! Initialize a character string as usual, and append the null later.
  character*5 message2 /'world'/

! Pass both strings to a C function that takes 2 (char *) arguments.
  call cfunc(%ref(message1), %ref(message2 // '\0'))
end
```

For compatibility with C language usage, you can encode the following escape sequences in XL Fortran character strings:

Table 18. Escape sequences for character strings

Escape	Meaning
\b	Backspace
\f	Form feed
\n	New-line
\r	Carriage return
\t	Tab
\0	Null
\'	Apostrophe (does not terminate a string)
\"	Double quotation mark (does not terminate a string)
\\	Backslash
\x	x, where x is any other character (the backslash is ignored)

If you do not want the backslash interpreted as an escape character within strings, you can compile with the **-qnoescape** option.

Passing arrays between languages

Fortran stores array elements in ascending storage units in column-major order. C stores array elements in row-major order. Fortran array indexes start at 1, while C array indexes start at 0.

The following example shows how a two-dimensional array that is declared by A(3,2) is stored in Fortran and C.

Table 19. Corresponding array layouts for Fortran and C. The Fortran array reference $A(X,Y,Z)$ can be expressed in C as $a[Z-1][Y-1][X-1]$. Keep in mind that although C passes individual scalar array elements by value, it passes arrays by reference.

	Fortran Element Name	C Element Name
Lowest storage unit	A(1,1)	A[0][0]
	A(2,1)	A[0][1]
	A(3,1)	A[1][0]
	A(1,2)	A[1][1]
	A(2,2)	A[2][0]
Highest storage unit	A(3,2)	A[2][1]

To pass all or part of a Fortran array to another language, you can use Fortran 90/Fortran 95 array notation:

```
REAL, DIMENSION(4,8) :: A, B(10)

! Pass an entire 4 x 8 array.
CALL CFUNC( A )
! Pass only the upper-left quadrant of the array.
CALL CFUNC( A(1:2,1:4) )
! Pass an array consisting of every third element of A.
CALL CFUNC( A(1:4:3,1:8) )
! Pass a 1-dimensional array consisting of elements 1, 2, and 4 of B.
CALL CFUNC( B( (/1,2,4/) ) )
```

Where necessary, the Fortran program constructs a temporary array and copies all the elements into contiguous storage. In all cases, the C routine needs to account for the column-major layout of the array.

Any array section or noncontiguous array is passed as the address of a contiguous temporary unless an explicit interface exists where the corresponding dummy argument is declared as an assumed-shape array or a pointer. To avoid the creation of array descriptors (which are not supported for interlanguage calls) when calling non-Fortran procedures with array arguments, either do not give the non-Fortran procedures any explicit interface, or do not declare the corresponding dummy arguments as assumed-shape or pointers in the interface:

```
! This explicit interface must be changed before the C function
! can be called.
INTERFACE
  FUNCTION CFUNC (ARRAY, PTR1, PTR2)
    INTEGER, DIMENSION (:) :: ARRAY      ! Change this : to *.
    INTEGER, POINTER, DIMENSION (:) :: PTR1 ! Change this : to *
                                           ! and remove the POINTER
                                           ! attribute.
    REAL, POINTER :: PTR2                ! Remove this POINTER
                                           ! attribute or change to TARGET.
  END FUNCTION
END INTERFACE
```

Passing pointers between languages

Integer **POINTERS** always represent the address of the pointee object and must be passed by value:

```
CALL CFUNC(%VAL(INTPTR))
```

Fortran 90 **POINTERS** can also be passed back and forth between languages but only if there is no explicit interface for the called procedure or if the argument in

the explicit interface does not have a **POINTER** attribute or assumed-shape declarator. You can remove any **POINTER** attribute or change it to **TARGET** and can change any deferred-shape array declarator to be explicit-shape or assumed-size.

Because of XL Fortran's call-by-reference conventions, you must pass even scalar values from another language as the address of the value, rather than the value itself. For example, a C function passing an integer value *x* to Fortran must pass `&x`. Also, a C function passing a pointer value *p* to Fortran so that Fortran can use it as an integer **POINTER** must declare it as `void **p`. A C array is an exception: you can pass it to Fortran without the `&` operator.

Passing arguments by reference or by value

To call subprograms written in languages other than Fortran (for example, user-written C programs, or operating system routines), the actual arguments may need to be passed by a method different from the default method used by Fortran. C routines, including those in system libraries such as **libc.so**, require you to pass arguments by value instead of by reference. (Although C passes individual scalar array elements by value, it passes arrays by reference.)

You can change the default passing method by using the **%VAL** 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 64-bit intermediate values.
- Of sequence-derived type, it is passed as multiple 32-bit intermediate values.

Byte-named constants and variables are passed as if they were **INTEGER(1)**. If the actual argument is a **CHARACTER(1)**, the compiler pads it on the left with zeros to a 32-bit value, regardless of whether you specified the **-qctyp1ss** compiler option.

In Linux 32-bit only, structures are actually copied, and then the address of the copy is passed by value.

In 64-bit mode

If the actual argument is one of the following:

- An integer or a logical that is shorter than 64 bits, it is sign-extended to a 64-bit value.
- Of type real or complex, it is passed as multiple 64-bit intermediate values.
- Of sequence-derived type, it is passed as multiple 64-bit intermediate values.

Byte-named constants and variables are passed as if they were **INTEGER(1)**. If the actual argument is a **CHARACTER(1)**, the compiler pads it on the left with zeros to a 64-bit value, regardless of whether you specified the **-qctyp1ss** compiler option.

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.

A dummy argument with the **VALUE** attribute can be of character type if you omit the length parameter or specify it using an initialization expression with a value of 1.

You must not specify the **VALUE** attribute with the following:

- Arrays
- Derived types with **ALLOCATABLE** components
- Dummy procedures

```
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
```

Passing complex values to/from gcc

Passing complex values between Fortran and Gnu C++ depends on what is specified for the **-qfloat=[no]complexgcc** suboption. If **-qfloat=complexgcc** is specified, the compiler uses Linux conventions when passing or returning complex numbers. **-qfloat=nocomplexgcc** is the default.

For **-qfloat=complexgcc** in 32-bit mode, the compiler passes **COMPLEX *8** values in 2 general-purpose registers (GPRs) and **COMPLEX *16** values in 4 GPRs. In 64-bit mode, **COMPLEX *8** values are passed in 1 GPR, and **COMPLEX *16** in 2 GPRs. For **-qfloat=nocomplexgcc**, **COMPLEX *8** and **COMPLEX *16** values are passed in 2 floating-point registers (FPRs). **COMPLEX *32** values are always passed in 4 FPRs for both **-qfloat=complexgcc** and **-qfloat=nocomplexgcc** (since **gcc** does not support **COMPLEX*32**).

For `-qfloat=complexgcc` in 32-bit mode, `COMPLEX *8` values are returned in GPR3-GPR4, and `COMPLEX *16` in GPR3-GPR6. In 64-bit mode, `COMPLEX *8` values are returned in GPR3, and `COMPLEX*16` in GPR 3-GPR4. For `-qfloat=nocomplexgcc`, `COMPLEX *8` and `COMPLEX *16` values are returned in FPR1-FPR2. For both `-qfloat=complexgcc` and `-qfloat=nocomplexgcc`, `COMPLEX *32` is always returned in FPR1-FPR4.

Returning values from Fortran functions

XL Fortran does not support calling certain types of Fortran functions from non-Fortran procedures. If a Fortran function returns a pointer, array, or character of nonconstant length, do not call it from outside Fortran.

You can call such a function indirectly:

```
SUBROUTINE MAT2(A,B,C)      ! You can call this subroutine from C, and the
                           ! result is stored in C.
INTEGER, DIMENSION(10,10) :: A,B,C
C = ARRAY_FUNC(A,B)        ! But you could not call ARRAY_FUNC directly.
END
```

Arguments with the OPTIONAL attribute

When you pass an optional argument by reference, the address in the argument list is zero if the argument is not present.

When you pass an optional argument by value, the value is zero if the argument is not present. The compiler uses an extra register argument to differentiate that value from a regular zero value. If the register has the value 1, the optional argument is present; if it has the value 0, the optional argument is not present.

Related information: See “Order of arguments in argument list” on page 209.

Assembler-level subroutine linkage conventions

The subroutine linkage convention specifies the machine state at subroutine entry and exit, allowing routines that are compiled separately in the same or different languages to be linked. The information on subroutine linkage and system calls in the *System V Application Binary Interface: PowerPC Processor Supplement* and *64-bit PowerPC ELF Application Binary Interface Supplement* are the base references on this topic. You should consult these for full details. This section summarizes the information needed to write mixed-language Fortran and assembler programs or to debug at the assembler level, where you need to be concerned with these kinds of low-level details.

The system linkage convention passes arguments in registers, taking full advantage of the large number of floating-point registers (FPRs), 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 20. Floating-point register usage across calls

Register	Preserved Across Calls	Use
0	no	

Table 20. Floating-point register usage across calls (continued)

Register	Preserved Across Calls	Use
1	no	FP parameter 1, function return 1.
2	no	FP parameter 2, function return 2.
3	no	FP parameter 3, function return complex *32.
4	no	FP parameter 4, function return complex *32.
⋮	⋮	⋮
8	no	FP parameter 8
9-13	no	
14-31	yes	local variables

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

Table 21. General-purpose register usage across calls

Register	Preserved Across Calls	Use
0	no	
1	yes	Stack pointer.
2	yes	System-reserved.
3	no	1st word of arg list; return value 1.
4	no	2nd word of arg list; return value 2.
5	no	3rd word of arg list
⋮	⋮	⋮
10	no	8th word of arg list.
11-12	no	
13	yes	SDA pointer.
14-30	no	Local variables.
31	yes	Local variables or “environment pointers”.

If a register is not designated as preserved, its contents may be changed during the call, and the caller is responsible for saving any registers whose values are needed later. Conversely, if a register is supposed to be preserved, the callee is responsible for preserving its contents across the call, and the caller does not need any special action.

The following table lists special-purpose register conventions.

Table 22. Special-purpose register usage across calls

Register	Preserved Across Calls
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

Table 22. Special-purpose register usage across calls (continued)

Register	Preserved Across Calls
XER register	no
FPSCR register	no

The stack

The stack is a portion of storage that is used to hold local storage, register save areas, parameter lists, and call-chain data. The stack grows from higher addresses to lower addresses. A stack pointer register (register 1) is used to mark the current “top” of the stack.

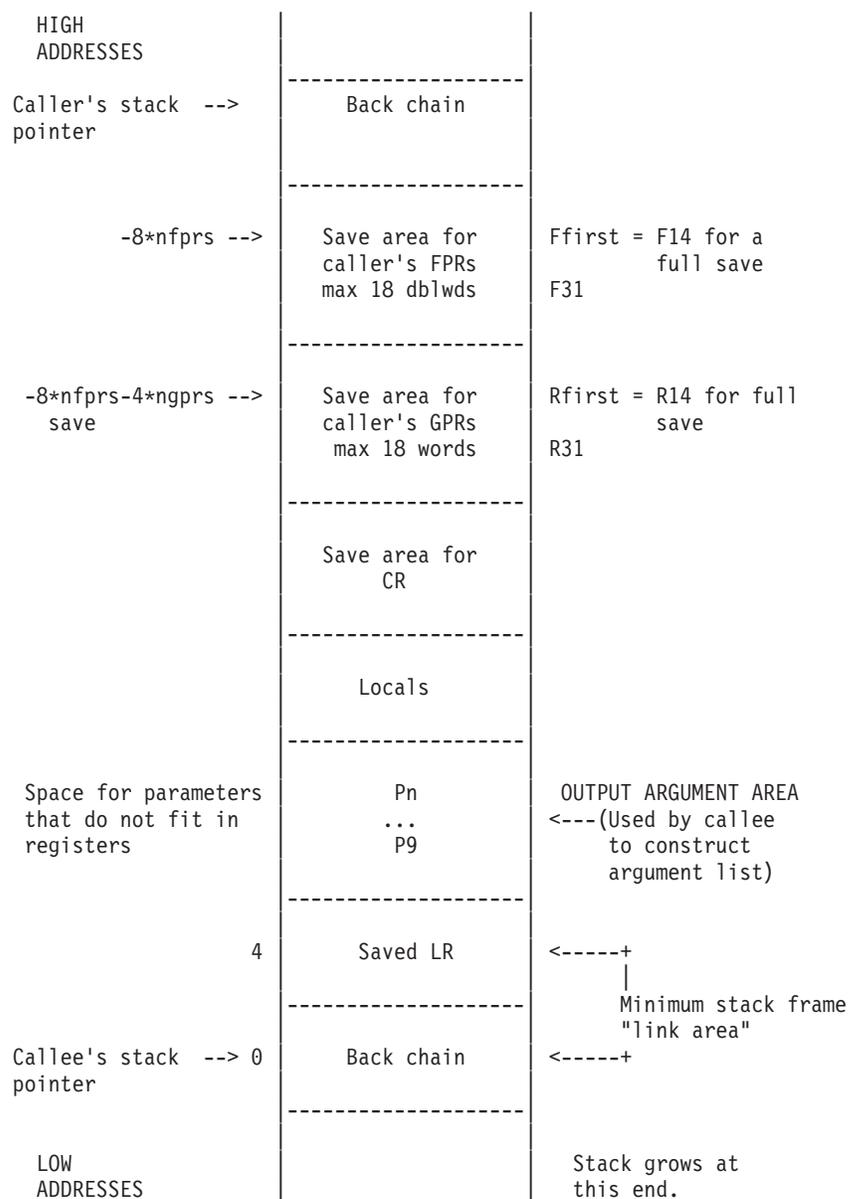
A stack frame is the portion of the stack that is used by a single procedure. The input parameters are considered part of the current stack frame. In a sense, each output argument belongs to both the caller’s and the callee’s stack frames. In either case, the stack frame size is best defined as the difference between the caller’s stack pointer and the callee’s.

The following diagrams show the storage maps of typical stack frames for 32-bit and 64-bit environments.

In these diagrams, the current routine has acquired a stack frame that allows it to call other functions. If the routine does not make any calls and there are no local variables or temporaries, and it does not need to save any non-volatile registers, the function need not allocate a stack frame. It can still use the register save area at the top of the caller’s stack frame, if needed.

The stack frame is double-word aligned.

Runtime Stack for 32-bit Environment



Runtime Stack for 64-bit Environment

Low Addresses		Stack grows at this end.
Callee's stack pointer --> 0	Back chain	
8	Saved CR	
16	Saved LR	
24-32	Reserved	<--- LINK AREA
40	Saved TOC	(callee)
Space for P1-P8 is always reserved	P1	OUTPUT ARGUMENT AREA
	...	<---(Used by callee
	Pn	to construct
		argument list)
	Callee's stack area	<--- LOCAL STACK AREA
		(Possible word wasted for alignment.)
-8* <i>nfprs</i> -8* <i>ngprs</i> -->	Save area for caller's GPR max 19 doublewords	Rfirst = R13 for full save R31
-8* <i>nfprs</i> -->	Save area for caller's FPR max 18 dblwds	Ffirst = F14 for a full save F31
Caller's stack pointer --> 0	Back chain	
8	Saved CR	
16	Saved LR	
24-32	Reserved	<--- LINK AREA
40	Saved TOC	(caller)
Space for P1-P8 is always reserved	P1	INPUT PARAMETER AREA
	...	<---(Callee's input
	Pn	parameters found
		here. Is also
	Callee's stack area	caller's arg area.)
High Addresses		

The Link Area and Minimum Stack Frame

In a 32-bit environment, the link area consists of two words at offset zero from the callee's stack pointer on entry to a procedure. The first word contains the caller's back chain (pointer to the previous stack frame). The second word is the location where the caller saves the Link Register (LR), if it is needed.

In a 64-bit environment, this area consists of six doublewords at offset zero from the caller's stack pointer on entry to a procedure. The first doubleword contains the caller's back chain (stack pointer). The second doubleword is the location where the callee saves the Condition Register (CR) if it is needed. The third doubleword is the location where the callee's prolog code saves the Link Register if it is needed. The fourth doubleword is reserved for C **SETJMP** and **LONGJMP** processing, and the fifth doubleword is reserved for future use. The last doubleword (doubleword 6) is reserved for use by the global linkage routines that are used when calling routines in other object modules (for example, in shared libraries).

The input parameter area

In a 32-bit environment, the input parameters that do not fit in registers go into the output argument area (P9... Pn).

In a 64-bit environment, the input parameter area is a contiguous piece of storage reserved by the calling program to represent the register image of the input parameters of the callee. The input parameter area is double-word aligned and is located on the stack directly following the caller's link area. This area is at least 8 doublewords in size. If more than 8 doublewords of parameters are expected, they are stored as register images that start at positive offset 112 from the incoming stack pointer.

The first 8 doublewords only appear in registers at the call point, never in the stack. Remaining words are always in the stack, and they can also be in registers.

The register save area

In a 32-bit environment, the register save area provides the space that is needed to save all nonvolatile FPRs and GPRs used by the callee program. The FPRs are saved next to caller's minimum stack frame. The GPRs are saved below the FPRs (in lower addresses).

In a 64-bit environment, the register save area is double-word aligned. It provides the space that is needed to save all nonvolatile FPRs and GPRs used by the callee program. The FPRs are saved next to the link area. The GPRs are saved below the FPRs (in lower addresses). The called function may save the registers here even if it does not need to allocate a new stack frame. The system-defined stack floor includes the maximum possible save area:

32-bit platforms: 18*8 for FPRs + 18*4 for GPRs
64-bit platforms: 18*8 for FPRs + 19*8 for GPRs

A callee needs only to save the nonvolatile registers that it actually uses.

The local stack area

The local stack area is the space that is allocated by the callee procedure for local variables and temporaries.

The output parameter area

In a 32-bit environment, the input parameters that do not fit in registers go into the output argument area (P9... Pn).

If more than 8 words are being passed, an extension list is constructed beginning at offset 8 from the current stack pointer.

The first 8 words only appear in registers at the call point, never in the stack. Remaining words are always in the stack, and they can also be in registers.

In a 64-bit environment, the output parameter area (P1...Pn) must be large enough to hold the largest parameter list of all procedures that the procedure that owns this stack frame calls. This area is at least 8 doublewords long, regardless of the length or existence of any argument list. If more than 8 doublewords are being passed, an extension list is constructed, which begins at offset 112 from the current stack pointer.

The first 8 doublewords only appear in registers at the call point, never in the stack. Remaining doublewords are always in the stack, and they can also be in registers.

Linkage convention for argument passing

The system linkage convention takes advantage of the large number of registers available. The linkage convention passes arguments in both GPRs and FPRs. Two fixed lists, R3-R10 and FP1-FP13, specify the GPRs and FPRs available for argument passing.

When there are more argument words than available argument GPRs and FPRs, the remaining words are passed in storage on the stack. The values in storage are the same as if they were in registers.

In a 64-bit environment, the size of the parameter area is sufficient to contain all the arguments passed on any call statement from a procedure that is associated with the stack frame. Although not all the arguments for a particular call actually appear in storage, it is convenient to consider them as forming a list in this area, each one occupying one or more words.

For call by reference (as is the default for Fortran), the address of the argument is passed in a register. The following information refers to call by value, as in C or as in Fortran when `%VAL` is used. For purposes of their appearance in the list, arguments are classified as floating-point values or non-floating-point values:

In a 32-bit Environment

- Each **INTEGER(8)** and **LOGICAL(8)** argument requires two words.
- Any other non-floating-point scalar argument of intrinsic type or procedure/function pointers requires one word and appears in that word exactly as it would appear in a GPR. It is signed or unsigned/extended, if language semantics specify, and is word aligned.
- Each single-precision (**REAL(4)**) value occupies one word. Each double-precision (**REAL(8)**) value occupies two successive words in the list. Each extended-precision (**REAL(16)**) value occupies four successive words in the list.
- A **COMPLEX** value occupies twice as many words as a **REAL** value with the same kind type parameter.
- In Fortran and C, structure values are passed “val-by-ref”. That is, the compiler actually passes the address of a copy of the structure.

In a 64-bit environment

- All non-floating-point values require one doubleword that is doubleword aligned.
- Each single-precision (**REAL(4)**) value and each double-precision (**REAL(8)**) value occupies one doubleword in the list. Each extended-precision (**REAL(16)**) value occupies two successive doublewords in the list.
- A **COMPLEX** value occupies twice as many doublewords as a **REAL** value with the same kind type parameter.
- In Fortran and C, structure values appear in successive words as they would anywhere in storage, satisfying all appropriate alignment requirements. Structures are aligned to a doubleword and occupy $(\text{sizeof}(\text{struct } X)+7)/8$ doublewords, with any padding at the end. A structure that is smaller than a doubleword is left-justified within its doubleword or register. Larger structures can occupy multiple registers and may be passed partly in storage and partly in registers.
- Other aggregate values 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 210 for more information.)

Argument passing rules (by value)

From the following illustration, we state these rules:

- In a 32-bit environment, arguments to called functions are passed in the GPRs and FPRs. Up to eight words are passed in GPR3-GPR10 and up to eight floating-point arguments in FPR1-FPR8. If fewer arguments are passed, unneeded registers are not loaded. If the passed arguments will not fit in registers, only enough space to hold the arguments that do not fit is allocated in the stack frame.
- In a 64-bit environment, if the called procedure treats the parameter list as a contiguous piece of storage (for example, if the address of a parameter is taken in C), the parameter registers are stored in the space reserved for them in the stack.
- A register image is stored on the stack.
- In a 64-bit environment, the argument area ($P_1\dots P_n$) must be large enough to hold the largest parameter list.

Here is an example of a call to a function:

```
f(%val(11), %val(12), %val(13), %val(14), %val(15), %val(16), %val(17),  
  %val(d1), %val(f1), %val(c1), %val(d2), %val(s1), %val(cx2))
```

where:

- l denotes integer(4) (fullword integer)
- d denotes real(8) (double precision)
- f denotes real(4) (real)
- s denotes integer(2) (halfword integer)
- c denotes character (one character)
- cx denotes complex(8) (double complex)

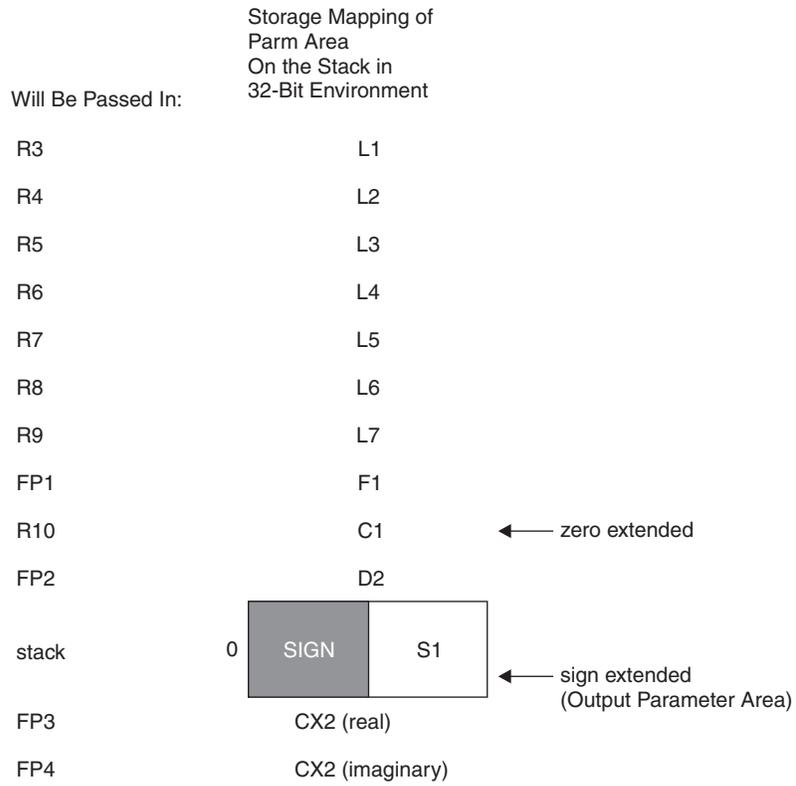


Figure 10. Storage mapping of parm area on the stack in 32-bit environment

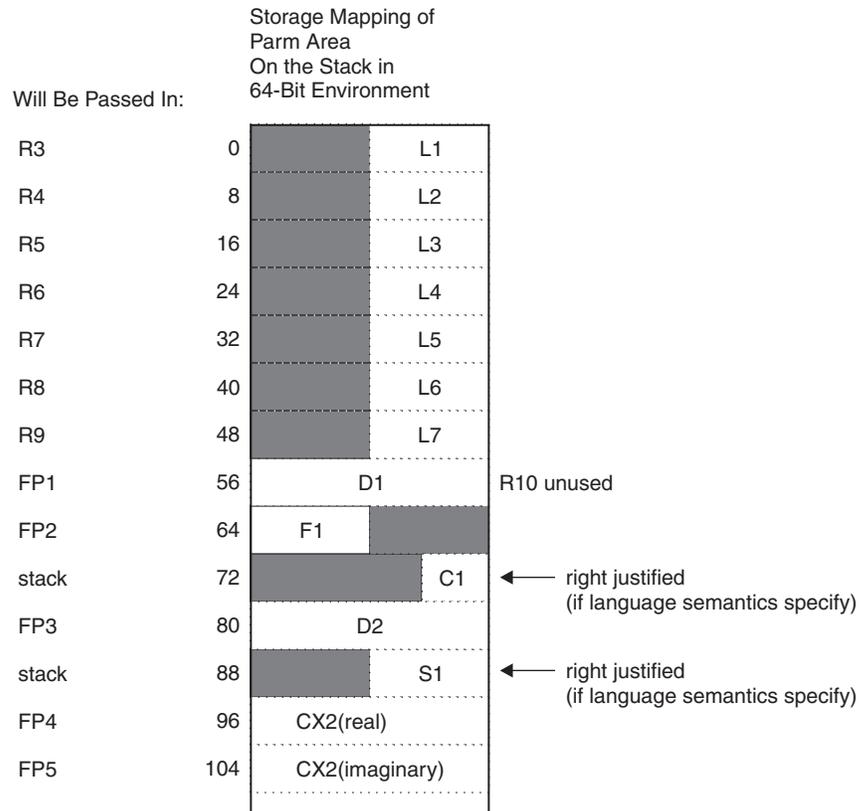


Figure 11. Storage mapping of parm area on the stack in 64-bit environment

Order of arguments in argument list

The argument list is constructed in the following order. Items in the same bullet appear in the same order as in the procedure declaration, whether or not argument keywords are used in the call.

- All addresses or values (or both) of actual arguments ¹
- "Present" indicators for optional arguments
- Length arguments for strings ¹

Linkage convention for function calls

In 64-bit mode, a routine has two symbols associated with it: a function descriptor (*name*) and an entry point (*.name*). When a call is made to a routine, the program branches to the entry point directly. Excluding the loading of parameters (if any) in the proper registers, compilers expand calls to functions (if any) 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*

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.

into the object module. Also, if a **NOP** instruction (`ORI R0,R0,0x0000`) immediately follows the **BL** instruction, the linker replaces the **NOP** instruction with the **LOAD** instruction `L R2, 20(R1)`.

2. If `foo` is bound in the same object module as its caller and a **LOAD** instruction `L R2,20(R1)` for 32-bit and `L R2,40(R1)` for 64-bit, or `ORI R0,R0,0` immediately follows the **BL** instruction, the linker replaces the **LOAD** instruction with a **NOP** (`ORI R0,R0,0`).

Note: For any export, the linker inserts the procedure's descriptor into the object module.

Pointers to functions

In 64-bit mode, a function pointer is a data type whose values range over procedure names. Variables of this type appear in several programming languages, such as C and Fortran. In Fortran, a dummy argument that appears in an **EXTERNAL** statement is a function pointer. Fortran provides support for the use of function pointers in contexts such as the target of a call statement or an actual argument of such a statement.

A function pointer is a fullword quantity that is the address of a function descriptor. The function descriptor is a 3-word object. The first word contains the address of the entry point of the procedure. The second has the address of the TOC of the object module in which the procedure is bound. The third is the environment pointer for some non-Fortran languages. There is only one function descriptor per entry point. It is bound into the same object module as the function it identifies if the function is external. The descriptor has an external name, which is the same as the function name but with a different storage class that uniquely identifies it. This descriptor name is used in all import or export operations.

Function values

Functions return their values according to type:

- In 32-bit mode, **INTEGER** and **LOGICAL** of kind 1, 2, and 4 are returned (sign/zero extended) in R3.
- In 64-bit mode, **INTEGER** and **LOGICAL** of kind 1, 2, and 4 are returned (right justified) in R3.
- In 64-bit mode, **INTEGER** and **LOGICAL** of kind 8 are returned in R3.
- **REAL** *4 or *8 are returned in FP1. **REAL** *16 are returned in FP1 and FP2.
- **COMPLEX** *8 or *16 are returned in FP1 and FP2. **COMPLEX** *32 are returned in FP1-FP4.
- In 32-bit mode when `-qfloat=complexgcc` is specified, **COMPLEX***8 is returned in R3-R4 and **COMPLEX***16 in R3-R6. In 64-bit mode, **COMPLEX***8 is returned in R3 and **COMPLEX***16 in R3-R4.
- 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

In 64-bit mode, the stack floor is a system-defined address below which the stack cannot grow. All programs in the system must avoid accessing locations in the stack segment that are below the stack floor.

All programs must maintain other system invariants that are related to the stack:

- No data is saved or accessed from an address lower than the stack floor.
- The stack pointer is always valid. When the stack frame size is more than 32 767 bytes, you must take care to ensure that its value is changed in a single instruction. This step ensures that there is no timing window where a signal handler would either overlay the stack data or erroneously appear to overflow the stack segment.

Stack overflow

The linkage convention requires no explicit inline check for overflow. The operating system uses a storage protection mechanism to detect stores past the end of the stack segment.

Prolog and epilog

On entry to a procedure, you might have to do some or all of the following steps:

1. Save the link register.
2. If you use any of the CR bits 8-23 (CR2, CR3, CR4, CR5), save the CR.
3. Save any nonvolatile FPRs that are used by this procedure in the FPR save area.
4. Save all nonvolatile 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 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 8-23 of the CR if necessary.
7. If you saved any FPRs, restore them.
8. Return to caller.

Traceback

In 64-bit mode, the compiler supports the traceback mechanism, which symbolic debuggers need to unravel the call or return stack. Each object module has a traceback table in the text segment at the end of its code. This table contains information about the object module, including the type of object module, as well as stack frame and register information.

Related information: You can make the traceback table smaller or remove it entirely with the **-qbttable** option.

Chapter 9. Implementation details of XL Fortran Input/Output (I/O)

This section discusses XL Fortran support (through extensions and platform-specific details) for the Linux file system.

Related information: See the **-qposition** option in the *XL Fortran Compiler Reference* and “Mixed-language input and output” on page 190.

Implementation details of file formats

XL Fortran implements files in the following manner:

Sequential-access unformatted files:

An integer that contains the length of the record precedes and follows each record. The length of the integer is 4 bytes for 32-bit applications. 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 a Linux file and end at the 300th byte.

If the length of the record of a direct-access file is greater than the total amount of data you want to write to the record, XL Fortran pads the record on the right with blanks (X'20').

Stream-access unformatted files:

Unformatted stream files are viewed as a collection of file storage units. In XL Fortran, a file storage unit is one byte.

A file connected for unformatted stream access has the following properties:

- The first file storage unit has position 1. Each subsequent file storage unit has a position that is one greater than that of the preceding one.
- For a file that can be positioned, file storage units need not be read or written in the order of their position. Any file storage unit may be read from the file while it is connected to a unit, provided that the file

storage unit has been written since the file was created, and if a READ statement for the connection is permitted.

Stream-access formatted files:

A record file connected for formatted stream access has the following properties:

- Some file storage units may represent record markers. The record marker is the newline character (X'0A').
- The file will have a record structure in addition to the stream structure.
- The record structure is inferred from the record markers that are stored in the file.
- Records can have any length up to the internal limit allowed by XL Fortran (See *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

You can specify file names as either relative (such as **file**, **dir/file**, or **../file**) or absolute (such as **/file** or **/dir/file**). The maximum length of a file name (the full path name) is 4095 characters, even if you only specify a relative path name in the I/O statement. The maximum length of a file name with no path is 255 characters.

You must specify a valid file name in such places as the following:

- The **FILE=** specifier of the **OPEN** and **INQUIRE** statements
- **INCLUDE** lines

Related information: To specify a file whose location depends on an environment variable, you can use the **GET_ENVIRONMENT_VARIABLE** intrinsic procedure to retrieve the value of the environment variable:

```
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

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**:

```

WRITE (6,10) "This message goes to stdout."
CLOSE (6)
WRITE (6,10) "This message goes in the file fort.6."
PRINT *, "Output to * now also goes in fort.6."
10  FORMAT (A)
END

```

The **FORM=** specifier of implicitly connected files has the value **FORMATTED** before any **READ**, **WRITE**, or **PRINT** statement is performed on the unit. The first such statement on such a file determines the **FORM=** specifier from that point on: **FORMATTED** if the formatting of the statement is format-directed, list-directed, or namelist; and **UNFORMATTED** if the statement is unformatted.

Preconnected files also have **FORM='FORMATTED'**, **STATUS='OLD'**, and **ACTION='READWRITE'** as default specifier values.

The other properties of a preconnected or implicitly connected file are the default specifier values for the **OPEN** statement. These files always use sequential access.

If you want XL Fortran to use your own file instead of the **fort.n** file, you can either specify your file for that unit through an **OPEN** statement or create a symbolic link before running the application. In the following example, there is a symbolic link between **myfile** and **fort.10**:

```
ln -s myfile fort.10
```

When you run an application that uses the 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**:

```
rm fort.10
```

File positioning

Table 23. Position of the file pointer when a file is opened with no **POSITION=** specifier

-qposition suboptions	Implicit OPEN		Explicit OPEN					
			STATUS = 'NEW'		STATUS = 'OLD'		STATUS = 'UNKNOWN'	
	File exists	File does not exist	File exists	File does not exist	File exists	File does not exist	File exists	File does not exist
option not specified	Start	Start	Error	Start	Start	Error	Start	Start

Table 23. Position of the file pointer when a file is opened with no POSITION= specifier (continued)

-qposition suboptions	Implicit OPEN		Explicit OPEN					
			STATUS = 'NEW'		STATUS = 'OLD'		STATUS = 'UNKNOWN'	
	File exists	File does not exist	File exists	File does not exist	File exists	File does not exist	File exists	File does not exist
appendold	Start	Start	Error	Start	End	Error	Start	Start
appendunknown	Start	Start	Error	Start	Start	Error	End	Start
appendold and appendunknown	Start	Start	Error	Start	End	Error	End	Start

I/O Redirection

You can use the redirection operator on the command line to redirect input to and output from your XL Fortran program. How you specify and use this operator depends on which shell you are running. Here is a **bash** example:

```
$ cat redirect.f
    write (6,*) 'This goes to standard output'
    write (0,*) 'This goes to standard error'
    read (5,*) i
    print *,i
end

$ xlf95 redirect.f
**_main    === End of Compilation 1 ===
1501-510  Compilation successful for file redirect.f.
$ # No redirection. Input comes from the terminal. Output goes to
$ # the screen.
$ a.out
This goes to standard output
This goes to standard error
4
4
$ # Create an input file.
$ echo >stdin 2
$ # Redirect each standard I/O stream.
$ a.out >stdout 2>stderr <stdin
$ cat stdout
This goes to standard output
2
$ cat stderr
This goes to standard error
```

Refer to your man pages for more information on redirection.

How 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.

Do not specify **ACTION='READWRITE'** for a pipe.

Do not use the **BACKSPACE** statement on files that are pseudo-devices or character special files.

Do not use the **REWIND** statement on files that are pseudo-devices or pipes.

Default record lengths

If a pseudo-device, pipe, or character special file is connected for formatted or unformatted sequential access with no **RECL=** specifier, or for formatted stream access, the default record length is 32 768 rather than 2 147 483 647, which is the default for sequential-access files connected to random-access devices. (See the **default_recl** runtime option.)

In certain cases, the default maximum record length for formatted files is larger, to accommodate programs that write long records to standard output. If a unit is connected to a terminal for formatted sequential access and there is no explicit **RECL=** qualifier in the **OPEN** statement, the program uses a maximum record length of 2 147 483 646 ($2^{31}-2$) bytes, rather than the usual default of 32 768 bytes. When the maximum record length is larger, formatted I/O has one restriction: **WRITE** statements that use the **T** or **TL** edit descriptors must not write more than 32 768 bytes. This is because the unit's internal buffer is flushed each 32 768 bytes, and the **T** or **TL** edit descriptors will not be able to move back past this boundary.

File permissions

A file must have the appropriate permissions (read, write, or both) for the corresponding operation being performed on it.

When a file is created, the default permissions (if the **umask** setting is 000) are both read and write for user, group, and other. You can turn off individual permission bits by changing the **umask** setting before you run the program.

Selecting error messages and recovery actions

By default, an 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 **XLFRTEOPTS** 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 **XLFRTEOPTS** setting.
- To make the program stop issuing messages each time it encounters an error, include **xrf_messages=no**.
- To disallow XL Fortran extensions to Fortran 90 at run time, include **langlvl=90std**. To disallow XL Fortran extensions to Fortran 95 at run time, include **langlvl=95std**. To disallow XL Fortran extensions to Fortran 2003

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:

```
# Switch defaults for some runtime settings.
XLFRT_OPTS="err_recovery=no:cnvrr=no"
export XLFRT_OPTS
```

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 `SETRT_OPTS` procedure:

```
PROGRAM RT_OPTS
USE XLFUTILITY
CALL SETRT_OPTS("err_recovery=no") ! Change setting.
... some I/O statements ...
CALL SETRT_OPTS("err_recovery=yes") ! Change it back.
... some more I/O statements ...
END
```

Because a user can change these settings through the `XLFRT_OPTS` environment variable, be sure to use `SETRT_OPTS` 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:

```
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: See “Mixed-language input and output” on page 190 and the `FLUSH` statement in the *XL Fortran Language Reference*.

Choosing locations and names for Input/Output files

If you need to override the default locations and names for input/output files, you can use the following methods without making any changes to the source code.

Naming files that are connected with no explicit name

To give a specific name to a file that would usually have a name of the form `fort.unit`, you must set the 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:

```
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)
XLFRT_OPTS="unit_vars=yes"      # Allow overriding default names.
XLFUNIT_1="/tmp/molecules.dat" # Use this named file.
XLFUNIT_12="./data/scratch"    # Relative to current directory.
XLFUNIT_123="/home/user/data"  # Somewhere besides /tmp.
export XLFRT_OPTS XLFUNIT_1 XLFUNIT_12 XLFUNIT_123
```

Notes:

1. The `XLFUNIT_number` variable name must be in uppercase, and *number* must not have any leading zeros.
2. `unit_vars=yes` might be only part of the value for the `XLFRT_OPTS` 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 `XLFRT_OPTS` 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:

```
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)
XLFRT_OPTS="scratch_vars=yes"    # Turn on scratch file naming.
XLFSCRATCH_1="/tmp/molecules.dat" # Use this named file.
XLFSCRATCH_12="./data/scratch"   # Relative to current directory.
XLFSCRATCH_123="/home/user/data" # Somewhere besides /tmp.
export XLFRT_OPTS XLFSCRATCH_1 XLFSCRATCH_12 XLFSCRATCH_123
```

Notes:

1. The `XLFSCRATCH_number` variable name must be in uppercase, and *number* must not have any leading zeros.
2. `scratch_vars=yes` might be only part of the value for the `XLFRTEOPTS` variable, depending on what other 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 `XLFSCRATCH_number` variable is not set when the program is run, the program chooses a unique file name for the scratch file and puts it in the directory named by the `TMPDIR` variable or in the `/tmp` directory if the `TMPDIR` variable is not set.

Asynchronous I/O

You may need to use asynchronous I/O for speed and efficiency in scientific programs that perform I/O for large amounts of data. Synchronous I/O blocks the execution of an application until the I/O operation completes. Asynchronous I/O allows an application to continue processing while the I/O operation is performed in the background. You can modify applications to take advantage of the ability to overlap processing and I/O operations. Multiple asynchronous I/O operations can also be performed simultaneously on multiple files that reside on independent devices. For a complete description of the syntax and language elements that you require to use this feature, see the *XL Fortran Language Reference* under the topics:

- **INQUIRE** Statement
- **OPEN** Statement
- **READ** Statement
- **WAIT** Statement
- **WRITE** Statement

Execution of an asynchronous data transfer operation

The effect of executing an asynchronous data transfer operation will be as if the following steps were performed in the order specified, with steps (6)-(9) possibly occurring asynchronously:

1. Determine the direction of the data transfer.
2. Identify the unit.
3. Establish the format if one is present.
4. Determine whether an error condition, end-of-file condition, or end-of-record condition has occurred.
5. Cause the variable that you specified in the `IOSTAT=` specifier in the data transfer statement to become defined.
6. Position the file before you transfer data.
7. Transfer data between the file and the entities that you specified by the input/output list (if any).
8. Determine whether an error condition, end-of-file condition, or end-of-record condition has occurred.
9. Position the file after you transfer data.
10. Cause any variables that you specified in the `IOSTAT=` and `SIZE=` specifiers in the `WAIT` statement to become defined.

Usage

You can use Fortran asynchronous **READ** and **WRITE** statements to initiate asynchronous data transfers in Fortran. Execution continues after the asynchronous I/O statement, regardless of whether the actual data transfer has completed.

A program may synchronize itself with a previously initiated asynchronous I/O statement by using a **WAIT** statement. There are two forms of the **WAIT** statement:

1. In a **WAIT** statement without the **DONE=** specifier, the **WAIT** statement halts execution until the corresponding asynchronous I/O statement has completed:

```
integer idvar
integer, dimension(1000):: a
....
READ(unit_number, ID=idvar) a
....
WAIT(ID=idvar)
....
```

2. In a **WAIT** statement with the **DONE=** specifier, the **WAIT** statement returns the completion status of an asynchronous I/O statement:

```
integer idvar
logical done
integer, dimension(1000):: a
....
READ(unit_number, ID=idvar) a
....
WAIT(ID=idvar, DONE=done)
....
```

The variable you specified in the **DONE=** specifier is set to "true" if the corresponding asynchronous I/O statement completes. Otherwise, it is set to "false".

The actual data transfer can take place in the following cases:

- During the asynchronous **READ** or **WRITE** statement
- At any time before the execution of the corresponding **WAIT** statement
- During the corresponding **WAIT** statement

Because of the nature of asynchronous I/O, the actual completion time of the request cannot be predicted.

You specify Fortran asynchronous **READ** and **WRITE** statements by using the **ID=** specifier. The value set for the **ID=** specifier by an asynchronous **READ** or **WRITE** statement must be the same value specified in the **ID=** specifier in the corresponding **WAIT** statement. You must preserve this value until the associated asynchronous I/O statement has completed.

The following program shows a valid asynchronous **WRITE** statement:

```
program sample0
integer, dimension(1000):: a
integer idvar
a = ((i,i=1,1000)/)
WRITE(10, ID=idvar) a
WAIT(ID=idvar)
end
```

The following program is not valid, because XL Fortran destroys the value of the asynchronous I/O identifier before the associated **WAIT** statement:

```
program sample1
integer, dimension(1000):: a
integer idvar
a = ((i,i=1,1000)/)
WRITE(10, ID=idvar) a
idvar = 999 ! Valid id is destroyed.
WAIT(ID=idvar)
end
```

An application that uses asynchronous I/O typically improves performance by overlapping processing with I/O operations. The following is a simple example:

```

program sample2
integer (kind=4), parameter :: isize=1000000, icol=5
integer (kind=4) :: i, j, k
integer (kind=4), dimension(icol) :: handle
integer (kind=4), dimension(isize,icol), static :: a, a1

!
!   Opens the file for both synchronous and asynchronous I/O.
!
open(20,form="unformatted",access="direct", &
     status="scratch", recl=isize*4,asynch="yes")

!
!   This loop overlaps the initialization of a(:,j) with
!   asynchronous write statements.
!
!   NOTE: The array is written out one column at a time.
!         Since the arrays in Fortran are arranged in column
!         major order, each WRITE statement writes out a
!         contiguous block of the array.
!
do 200 j = 1, icol
  a(:,j) = (/ (i*j,i=1,isize) /)
  write(20, id=handle(j), rec=j) a(:,j)
200 end do

!
!   Wait for all writes to complete before reading.
!
do 300 j = 1, icol
  wait(id=handle(j))
300 end do

!
!   Reads in the first record.
!
read(20, id=handle(1), rec=1) a1(:,1)

do 400 j = 2, icol
  k = j - 1

!
!   Waits for a previously initiated read to complete.
!
  wait(id=handle(k))

!
!   Initiates the next read immediately.
!
  read(20, id=handle(j), rec=j) a1(:,j)

!
!   While the next read is going on, we do some processing here.
!
  do 350 i = 1, isize
    if (a(i,k) .ne. a1(i,k)) then
      print *, "(i,k) &
        & expected ", a(i,k), " got ", a1(i,k)
    end if
350   end do
400 end do

!
!   Finish the last record.
!
wait(id=handle(icol))

```

```

do 450 i = 1, isize
  if (a(i,icol) .ne. a1(i,icol)) then
    print *, "(,i,",",icol,)" &
      & expected ", a(i,icol), " got ", a1(i,icol)
  end if
450 end do

close(20)
end

```

Performance

To maximize the benefits of asynchronous I/O, you should only use it for large contiguous data items.

It is possible to perform asynchronous I/O on a large number of small items, but the overall performance will suffer. This is because extra processing overhead is required to maintain each item for asynchronous I/O. Performing asynchronous I/O on a larger number of small items is strongly discouraged. The following are two examples:

1. WRITE(unit_number, ID=idvar) a1(1:100000000:2)
2. WRITE(unit_number, ID=idvar) (a2(i,j),j=1,100000000)

Performing asynchronous I/O on unformatted sequential files is less efficient. This is because each record might have a different length, and these lengths are stored with the records themselves. You should use unformatted direct access or unformatted stream access, if possible, to maximize the benefits of asynchronous I/O.

Compiler-generated temporary I/O items

There are situations when the compiler must generate a temporary variable to hold the result of an I/O item expression. In such cases, synchronous I/O is performed on the temporary variable, regardless of the mode of transfer that you specified in the I/O statement. The following are examples of such cases:

1. For **READ**, when an array with vector subscripts appears as an input item:

```

a.      integer a(5), b(3)

        b = (/1,3,5/)
        read(99, id=i) a(b)
b.      real a(10)
        read(99, id=i) a(/1,3,5/)

```

2. For **WRITE**, when an output item is an expression that is a constant or a constant of certain derived types:

```

a.      write(99, id=i) 1000
b.      integer a
        parameter(a=1000)

        write(99, id=i) a
c.      type mytype
        integer a
        integer b
        end type mytype

        write(99, id=i) mytype(4,5)

```

3. For **WRITE**, when an output item is a temporary variable:

```

a.      write(99, id=i) 99+100
b.      write(99, id=i) a+b

```

```
c.      external ff
        real(8) ff
```

```
        write(99,id=i) ff()
```

4. For **WRITE**, when an output item is an expression that is an array constructor:

```
        write(99,id=i) (/1,2,3,4,5/)
```

5. For **WRITE**, when an output item is an expression that is a scalarized array:

```
        integer a(5),b(5)
        write(99,id=i) a+b
```

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.so** provides support for parallel execution of Fortran I/O statements.

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” on page 225 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” on page 227 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:

```
do i = 1, 10
  write(4, '(i4)', rec = i) a(i)
enddo
```

- Each thread performs I/O on a different part of a stream-access file. Different I/O statements cannot use the same, or overlapping, areas of a file.

```
do i = 1, 9
  write(4, '(i4)', pos = 1 + 5 * (i - 1)) a(i)
  ! We use 5 above because i4 takes 4 file storage
  ! units + 1 file storage unit for the record marker.
enddo
```

- In the case that each thread operates on a different file, since threads share the status of the logical units connected to the files, the thread still needs to obtain the lock on the logical unit for either retrieving or updating the status of the logical unit. However, the 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 = 11, 20
  write(i, '(i4)') a(i - 10)
enddo

```

For multiple threads to write to or read from the same sequential-access file, or to write to or read from the same stream-access file without using the **POS=** specifier, the order of records written out or read in depends on the order in which the threads execute the I/O statement on them. This order, as stated previously, is not predictable. Therefore, the result of an application could be incorrect if it assumes records are sequentially related and cannot be arbitrarily written out or read in. For example, if the following loop is parallelized, the numbers printed out will no longer be in the sequential order from 1 to 500 as the result of a serial execution:

```

do i = 1, 500
  print *, i
enddo

```

Applications that depend on numbers being strictly in the specified order will not work correctly.

The XL Fortran 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 section answers some common questions about floating-point processing, such as:

- How can I get predictable, consistent results?
- How can I get the fastest or the most accurate results?
- How can I detect, and possibly recover from, exception conditions?
- Which compiler options can I use for floating-point calculations?

Related information: This section makes frequent reference to the compiler options that are grouped together in *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.

Note: The discussions of single-, double-, and extended-precision calculations in this section all refer to the default situation, with **-qrealsize=4** and no **-qautodbl** specified. If you change these settings, keep in mind that the size of a Fortran **REAL**, **DOUBLE PRECISION**, and so on may change, but single precision, double precision, and extended precision (in lowercase) still refer to 4-, 8-, and 16-byte entities respectively.

The information in this section relates to floating-point processing on the PowerPC family of processors.

IEEE Floating-point overview

Here is a brief summary of the *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. 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 **rrm** among the **-qfloat** suboptions.

- If the data or program code contains signaling NaN values (NaN), include **nans** among the **-qfloat** suboptions. (A signaling NaN is different from a quiet NaN; you must explicitly code it into the program or data or create it by using the **-qinitauto** compiler option.)
- If you are compiling with **-O3**, or a higher base optimization level, include the **-qstrict** option also.

IEEE Single- and double-precision values

XL Fortran encodes single-precision and double-precision values in IEEE format. For the range and representation, see *Real* in the *XL Fortran Language Reference*.

IEEE Extended-precision values

The IEEE standard suggests, but does not mandate, a format for extended-precision values. XL Fortran does not use this format. “Extended-precision values” on page 233 describes the format that XL Fortran uses.

Infinites and NaNs

For single-precision real values:

- Positive infinity is represented by the bit pattern X'7F80 0000'.
- Negative infinity is represented by the bit pattern X'FF80 0000'.
- A signaling NaN is represented by any bit pattern between X'7F80 0001' and X'7FBF FFFF' or between X'FF80 0001' and X'FFBF FFFF'.
- A quiet NaN is represented by any bit pattern between X'7FC0 0000' and X'7FFF FFFF' or between X'FFC0 0000' and X'FFFF FFFF'.

For double-precision real values:

- Positive infinity is represented by the bit pattern X'7FF00000 00000000'.
- Negative infinity is represented by the bit pattern X'FFF00000 00000000'.
- A signaling NaN is represented by any bit pattern between X'7FF00000 00000001' and X'7FF7FFFF FFFFFFFF' or between X'FFF00000 00000001' and X'FFF7FFFF FFFFFFFF'.
- A quiet NaN is represented by any bit pattern between X'7FF80000 00000000' and X'7FFFFFFF FFFFFFFF' or between X'FFF80000 00000000' and X'FFFFFFF FFFFFFFF'.

These values do not correspond to any Fortran real constants. You can generate all of these by encoding the bit pattern directly, or by using the **ieee_value** function provided in the **ieee_arithmetic** module. Using the **ieee_value** function is the preferred programming technique, as it is allowed by the Fortran 2003 standard and the results are portable. Encoding the bit pattern directly could cause portability problems on machines using different bit patterns for the different values. All except signaling NaN values can occur as the result of arithmetic operations:

```

$ cat fp_values.f
real plus_inf, minus_inf, plus_nanq, minus_nanq, nans
real large

data plus_inf /z'7f800000'/
data minus_inf /z'ff800000'/
data plus_nanq /z'7fc00000'/
data minus_nanq /z'ffc00000'/
data nans /z'7f800001'/

print *, 'Special values:', plus_inf, minus_inf, plus_nanq, minus_nanq, nans

! They can also occur as the result of operations.
large = 10.0 ** 200
print *, 'Number too big for a REAL:', large * large
print *, 'Number divided by zero:', (-large) / 0.0
print *, 'Nonsensical results:', plus_inf - plus_inf, sqrt(-large)

! To find if something is a NaN, compare it to itself.
print *, 'Does a quiet NaN equal itself:', plus_nanq .eq. plus_nanq
print *, 'Does a signaling NaN equal itself:', nans .eq. nans
! Only for a NaN is this comparison false.

end
$ xlf95 -o fp_values fp_values.f
** _main === End of Compilation 1 ===
1501-510 Compilation successful for file fp_values.f.
$ fp_values
Special values: INF -INF NAN -NAN NAN
Number too big for a REAL: INF
Number divided by zero: -INF
Nonsensical results: NAN NAN
Does a quiet NaN equal itself: F
Does a signaling NaN equal itself: F

```

Exception-handling model

The IEEE standard defines several exception conditions that can occur:

OVERFLOW

The exponent of a value is too large to be represented.

UNDERFLOW

A nonzero value is so small that it cannot be represented without an extraordinary loss of accuracy. The value can be represented only as zero or a denormal number.

ZERODIVIDE

A finite nonzero value is divided by zero.

INVALID

Operations are performed on values for which the results are not defined. These include:

- Operations on signaling NaN values
- infinity - infinity
- 0.0 * infinity
- 0.0 / 0.0
- mod(x,y) or ieee_rem(x,y) (or other remainder functions) when x is infinite or y is zero
- The square root of a negative number
- Conversion of a floating point number to an integer when the converted value cannot be represented faithfully

- Comparisons involving NaN values

INEXACT

A computed value cannot be represented exactly, so a rounding error is introduced. (This exception is very common.)

XL Fortran always detects these exceptions when they occur, but 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 24. Results of IEEE exceptions, with and without trapping enabled

	Overflow	Underflow	Zerodivide	Invalid	Inexact
Exceptions not enabled (default)	INF	Denormalized number	INF	NaN	Rounded result
Exceptions enabled	Unnormalized number with biased exponent	Unnormalized number with biased exponent	No result	No result	Rounded result

Note: Because different results are possible, it is very important to make sure that any exceptions that are generated are handled correctly. See “Detecting and trapping floating-point exceptions” on page 238 for instructions on doing so.

Hardware-specific floating-point overview

Single- and double-precision values

The PowerPC floating-point hardware performs calculations in either IEEE single-precision (equivalent to **REAL(4)** in Fortran programs) or IEEE double-precision (equivalent to **REAL(8)** in Fortran programs).

Keep the following considerations in mind:

- Double precision provides greater range (approximately $10^{*(-308)}$ to 10^{*308}) and precision (about 15 decimal digits) than single precision (approximate range $10^{*(-38)}$ to 10^{*38} , with about 7 decimal digits of precision).
- Computations that mix single and double operands are performed in double precision, which requires conversion of the single-precision operands to double-precision. These conversions do not affect performance.
- Double-precision values that are converted to single-precision (such as when you specify the **SNGL** intrinsic or when a double-precision computation result is stored into a single-precision variable) require rounding operations. A rounding operation produces the correct single-precision value, which is based on the IEEE rounding mode in effect. The value may be less precise than the original double-precision value, as a result of rounding error. Conversions from double-precision values to single-precision values may reduce the performance of your code.
- Programs that manipulate large amounts of floating-point data may run faster if they use **REAL(4)** rather than **REAL(8)** variables. (You need to ensure that **REAL(4)** variables provide you with acceptable range and precision.) The programs may run faster because the smaller data size reduces memory traffic, which can be a performance bottleneck for some applications.

The floating-point hardware also provides a special set of double-precision operations that multiply two numbers and add a third number to the product. These combined multiply-add (**MAF**) operations are performed at the same speed 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 234 for the different ways you can control the rounding mode.

Programs that specify extended-precision values as hexadecimal, octal, binary, or Hollerith constants must follow these conventions:

- Extended-precision numbers are composed of two double-precision numbers with different magnitudes that do not overlap. That is, the binary exponents differ by at least the number of fraction bits in a **REAL(8)**. The high-order double-precision value (the one that comes first in storage) must have the larger magnitude. The value of the extended-precision number is the sum of the two double-precision values.
- For a value of NaN or infinity, you must encode one of these values within the high-order double-precision value. The low-order value is not significant.

Because an XL Fortran extended-precision value can be the sum of two values with greatly different exponents, leaving a number of assumed zeros in the fraction, the format actually has a variable precision with a minimum of about 31 decimal digits. You get more precision in cases where the exponents of the two double values differ in magnitude by more than the number of digits in a double-precision value. This encoding allows an efficient implementation intended for applications requiring more precision but no more range than double precision.

Notes:

1. In the discussions of rounding errors because of compile-time folding of expressions, keep in mind that this folding produces different results for extended-precision values more often than for other precisions.
2. Special numbers, such as NaN and infinity, are not fully supported for extended-precision values. Arithmetic operations do not necessarily propagate these numbers in extended precision.
3. XL Fortran does not always detect floating-point exception conditions (see “Detecting and trapping floating-point exceptions” on page 238) 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 237.

Selecting the rounding mode

To change the rounding mode in a program, you can call the **fpsets** and **fpgets** routines, which use an array of logicals named **fpstat**, defined in the include files **/opt/ibmcmp/xlf/11.1/include/fpdt.h** and **fpdc.h**. The **fpstat** array elements correspond to the bits in the floating-point status and control register.

For floating-point rounding control, the array elements **fpstat(fprn1)** and **fpstat(fprn2)** are set as specified in the following table:

Table 25. Rounding-mode bits to use with *fpsets* and *fpgets*

fpstat(fprn1)	fpstat(fprn2)	Rounding Mode Enabled
.true.	.true.	Round towards -infinity.
.true.	.false.	Round towards +infinity.
.false.	.true.	Round towards zero.
.false.	.false.	Round to nearest.

For example:

```
program fptest
  include 'fpdc.h'

  call fpgets( fpstat ) ! Get current register values.
  if ( (fpstat(fprn1) .eqv. .false.) .and. +
      (fpstat(fprn2) .eqv. .false.)) then
    print *, 'Before test: Rounding mode is towards nearest'
    print *, '          2.0 / 3.0 = ', 2.0 / 3.0
    print *, '          -2.0 / 3.0 = ', -2.0 / 3.0
  end if

  call fpgets( fpstat ) ! Get current register values.
  fpstat(fprn1) = .TRUE. ! These 2 lines mean round towards
  fpstat(fprn2) = .FALSE. ! +infinity.
  call fpsets( fpstat )
  r = 2.0 / 3.0
  print *, 'Round towards +infinity: 2.0 / 3.0= ', r

  call fpgets( fpstat ) ! Get current register values.
  fpstat(fprn1) = .TRUE. ! These 2 lines mean round towards
  fpstat(fprn2) = .TRUE. ! -infinity.
  call fpsets( fpstat )
  r = -2.0 / 3.0
  print *, 'Round towards -infinity: -2.0 / 3.0= ', r
end

! This block data program unit initializes the fpstat array, and so on.
```

```

        block data
        include 'fpdc.h'
        include 'fpdt.h'
    end

```

XL Fortran also provides several procedures that allow you to control the floating-point status and control register of the processor directly. These procedures are more efficient than the **fpsets** and **fpgets** subroutines because they are mapped into inlined machine instructions that manipulate the floating-point status and control register (fpscr) directly.

XL Fortran supplies the **get_round_mode()** and **set_round_mode()** procedures in the **xlf_fp_util** module. These procedures return and set the current floating-point rounding mode, respectively.

For example:

```

program fptest
  use, intrinsic :: xlf_fp_util
  integer(fpscr_kind) old_fpscr
  if ( get_round_mode() == fp_rnd_rn ) then
    print *, 'Before test: Rounding mode is towards nearest'
    print *, '          2.0 / 3.0 = ', 2.0 / 3.0
    print *, '          -2.0 / 3.0 = ', -2.0 / 3.0
  end if

  old_fpscr = set_round_mode( fp_rnd_rp )
  r = 2.0 / 3.0
  print *, 'Round towards +infinity: 2.0 / 3.0 = ', r

  old_fpscr = set_round_mode( fp_rnd_rm )
  r = -2.0 / 3.0
  print *, 'Round towards -infinity: -2.0 / 3.0 = ', r
end

```

XL Fortran supplies the **ieee_get_rounding_mode()** and **ieee_set_rounding_mode()** procedures in the **ieee_arithmetic** module. These portable procedures retrieve and set the current floating-point rounding mode, respectively.

For example:

```

program fptest
  use, intrinsic :: ieee_arithmetic
  type(ieee_round_type) current_mode
  call ieee_get_rounding_mode( current_mode )
  if ( current_mode == ieee_nearest ) then
    print *, 'Before test: Rounding mode is towards nearest'
    print *, '          2.0 / 3.0 = ', 2.0 / 3.0
    print *, '          -2.0 / 3.0 = ', -2.0 / 3.0
  end if

  call ieee_set_rounding_mode( ieee_up )
  r = 2.0 / 3.0
  print *, 'Round towards +infinity: 2.0 / 3.0 = ', r

  call ieee_set_rounding_mode( ieee_down )
  r = -2.0 / 3.0
  print *, 'Round towards -infinity: -2.0 / 3.0 = ', r
end

```

Notes:

1. Extended-precision floating-point values must only be used in round-to-nearest mode.

2. For thread-safety and reentrancy, the include file `/opt/ibmcmp/xlf/11.1/include/fpdc.h` contains a `THREADLOCAL` directive that is protected by the trigger constant `IBMT`. The invocation commands `xlf_r`, `xlf90_r`, `xlf95_r`, 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 `/opt/ibmcmp/xlf/11.1/include/fpdc.h` in code that is not intended to be thread-safe, do not specify `IBMT` as a trigger constant.

Minimizing rounding errors

There are several strategies for handling rounding errors and other unexpected, slight differences in calculated results. You may want to consider one or more of the following strategies:

- Minimizing the amount of overall rounding
- Delaying as much rounding as possible to run time
- Ensuring that if some rounding is performed in a mode other than round-to-nearest, *all* rounding is performed in the same mode

Minimizing overall rounding

Rounding operations, especially in loops, reduce code performance and may have a negative effect on the precision of computations. Consider using double-precision variables instead of single-precision variables when you store the temporary results of double-precision calculations, and delay rounding operations until the final result is computed.

Delaying rounding until run time

The compiler evaluates floating-point expressions during compilation when it can, so that the resulting program does not run more slowly due to unnecessary 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 234 with this command if the program consistently uses round-to-plus-infinity mode:

```
xlf95 -qieee=plus -qfloat=rrm changes_rounding_mode.f
```

Duplicating the floating-point results of other systems

The default **-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 **-qarch** settings while using new **-qarch** settings like **ppc** or **pwr6**, you should additionally use **-qfloat=nosingle** and **-qfloat=norndsngl**.

To duplicate the double-precision results of programs on systems with different floating-point architectures (without multiply-add instructions), specify the **nomaf** suboption of the **-qfloat** option. This suboption prevents the compiler from generating any multiply-add 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 **REAL** items is different from that on systems running XL Fortran, use the **-qrealsize** option to change the default **REAL** size when compiling with XL Fortran.

If the system whose results you want to duplicate preserves full double precision for default real constants that are assigned to **DOUBLE PRECISION** variables, use the **-qdpc** or **-qrealsize** option.

If results consistent with other systems are important to you, include **norsqrt** and **nofold** in the settings for the **-qfloat** option. If you specify the option **-O3**, include **-qstrict** too.

Maximizing floating-point performance

If performance is your primary concern and you want your program to be relatively safe but do not mind if results are slightly different (generally more precise) from what they would be otherwise, optimize the program with the **-O** option, and specify **-qfloat=rsqrt:hssngl:fltint**. The following section describes the functions of these suboptions:

- The **rsqrt** suboption replaces division by a square root with multiplication by the reciprocal of the root, a faster operation that may not produce precisely the same result.
- The **hssngl** suboption improves the performance of single-precision (**REAL(4)**) floating-point calculations by suppressing rounding operations that are required by the Fortran language but are not necessary for correct program execution. The results of floating-point expressions are kept in double precision where the original program would round them to single-precision. These results are then used in later expressions instead of the rounded results.

To detect single-precision floating-point overflows and underflows, rounding operations are still inserted when double-precision results are stored into single-precision memory locations. However, if optimization removes such a store operation, **hssngl** also removes the corresponding rounding operation, possibly preventing the exception. (Depending on the characteristics of your program, you may or may not care whether the exception happens.)

The **hssngl** suboption is safe for all types of programs because it always only *increases* the precision of floating-point calculations. Program results may differ because of the increased precision and because of avoidance of some exceptions.

- The **fltint** suboption speeds up float-to-integer conversions by reducing error checking for overflows. You should make sure that any floats that are converted to integers are not outside the range of the corresponding integer types.

Detecting and trapping floating-point exceptions

As stated earlier, the IEEE standard for floating-point arithmetic defines a number of exception (or error) conditions that might require special care to avoid or recover from. The following sections are intended to help you make your programs work safely in the presence of such exception conditions while sacrificing the minimum amount of performance.

The floating-point hardware always detects a number of floating-point exception conditions (which the IEEE standard rigorously defines): overflow, underflow, zerodivide, invalid, and inexact.

By default, the only action that occurs is that a status flag is set. The program continues without a problem (although the results from that point on may not be what you expect). If you want to know when an exception occurs, you can arrange for one or more of these exception conditions to generate a signal.

The signal causes a branch to a handler routine. The handler receives information about the type of signal and the state of the program when the signal occurred. It can produce a core dump, display a listing showing where the exception occurred, modify the results of the calculation, or carry out some other processing that you specify.

The XL Fortran compiler uses the operating system facilities for working with floating-point exception conditions. These facilities indicate the presence of floating-point exceptions by generating **SIGFPE** signals.

Compiler features for trapping floating-point exceptions

To turn on XL Fortran exception trapping, compile the program with the **-qfltrap** option and some combination of suboptions that includes **enable**. This option uses trap operations to detect floating-point exceptions and generates **SIGFPE** signals when exceptions occur, provided that a signal handler for **SIGFPE** is installed..

-qfltrap also has suboptions that correspond to the names of the exception conditions. For example, if you are only concerned with handling overflow and underflow exceptions, you could specify something similar to the following:

```
xlf95 -qfltrap=overflow:underflow:enable compute_pi.f
```

You only need **enable** when you are compiling the main program. However, it is very important and does not cause any problems if you specify it for other files, so always include it when you use **-qfltrap**.

An advantage of this approach is that performance impact is relatively low. To further reduce performance impact, you can include the **imprecise** suboption of the **-qfltrap** option. This suboption delays any trapping until the program reaches the start or end of a subprogram.

The disadvantages of this approach include the following:

- It only traps exceptions that occur in code that you compiled with **-qfltrap**, which does not include system library routines.

- It is generally not possible for a handler to substitute results for failed calculations if you use the **imprecise** suboption of **-qfltrap**.

Notes:

1. If your program depends on floating-point exceptions occurring for particular operations, also specify **-qfloat** suboptions that include **nofold**. Otherwise, the compiler might replace an exception-producing calculation with a constant NaN or infinity value, or it might eliminate an overflow in a single-precision operation.
2. The suboptions of the **-qfltrap** option replace an earlier technique that required you to modify your code with calls to the **fpsets** and **fpgets** procedures. You no longer require these calls for exception handling if you use the appropriate **-qfltrap** settings.

Attention: If your code contains **fpsets** calls that enable checking for floating-point exceptions and you do not use the **-qfltrap** option when compiling the whole program, the program will produce unexpected results if exceptions occur, as explained in Table 24 on page 232.

Installing an exception handler

When a program that uses the XL Fortran or Linux exception-detection facilities encounters an exception condition, it generates a signal. This causes a branch to whatever handler is specified by the program.

By default, Linux does not trap on floating-point exceptions unless a signal handler is installed. To produce a core file, you can use the **xl__trcedump** signal handler described below. If you want to install a **SIGTRAP** or **SIGFPE** signal handler, use the **-qsigtrap** option. It allows you to specify an XL Fortran handler that produces a traceback or to specify a handler you have written:

```
xlf95 -qfltrap=ov:und:en pi.f # No exceptions trapped
xlf95 -qfltrap=ov:und:en -qsigtrap=xl__trcedump pi.f # Uses the xl__trcedump handler
# to dump core on an exception
xlf95 -qfltrap=ov:und:en -qsigtrap=return_22_over_7 pi.f # Uses any other handler
```

You can also install an alternative exception handler, either one supplied by XL Fortran or one you have written yourself, by calling the **SIGNAL** subroutine (defined in **/opt/ibmcmp/xlf/11.1/include/fexcp.h**):

```
INCLUDE 'fexcp.h'
CALL SIGNAL(SIGTRAP,handler_name)
CALL SIGNAL(SIGFPE,handler_name)
```

The XL Fortran exception handlers and related routines are:

xl__ieee	Produces a traceback and an explanation of the signal and continues execution by supplying the default IEEE result for the failed computation. This handler allows the program to produce the same results as if exception detection was not turned on.
xl__trce	Produces a traceback and stops the program.
xl__trcedump	Produces a traceback and a core file and stops the program.
xl__sigdump	Provides a traceback that starts from the point at which it is called and provides information about the signal. You can only call it from inside a user-written signal handler. It does not stop the

program. To successfully continue, the signal handler must perform some cleanup after calling this subprogram.

xl__trbk

Provides a traceback that starts from the point at which it is called. You call it as a subroutine from your code, rather than specifying it with the **-qsigtrap** option. It requires no parameters. It does not stop the program.

All of these handler names contain double underscores to avoid duplicating names that you declared in your program. All of these routines work for both **SIGTRAP** and **SIGFPE** signals.

You can use the **-g** compiler option to get line numbers in the traceback listings. The file `/opt/ibmcmp/xlf/11.1/include/fsignal.h` defines a Fortran derived type similar to the `sigcontext` structure in the `signal.h` system header. You can write a Fortran signal handler that accesses this derived type.

Related information: “Sample programs for exception handling” on page 243 lists some sample programs that illustrate how to use these signal handlers or write your own. For more information, see the **SIGNAL** procedure in the *XL Fortran Language Reference*.

Producing a core file

To produce a core file, specify the `xl__trcedump` handler.

Controlling the floating-point status and control register

Before the **-qflttrap** suboptions or the **-qsigtrap** options, most of the processing for floating-point exceptions required you to change your source files to turn on exception trapping or install a signal handler. Although you can still do so, for any new applications, we recommend that you use the options instead.

To control exception handling at run time, compile without the **enable** suboption of the **-qflttrap** option:

```
xlf95 -qflttrap compute_pi.f # Check all exceptions, but do not trap.
xlf95 -qflttrap=ov compute_pi.f # Check one type, but do not trap.
```

Then, inside your program, manipulate the `fpstats` array (defined in the include file `/opt/ibmcmp/xlf/11.1/include/fpdc.h`) and call the `fpsets` subroutine to specify which exceptions should generate traps.

See the sample program that uses `fpsets` and `fpgets` in “Selecting the rounding mode” on page 234.

Another method is to use the `set_fpscr_flags()` subroutine in the `xlf_fp_util` module. This subroutine allows you to set the floating-point status and control register flags you specify in the **MASK** argument. Flags that you do not specify in **MASK** remain unaffected. **MASK** must be of type `INTEGER(FPSCR_KIND)`. For example:

```
USE, INTRINSIC :: xlf_fp_util
INTEGER(FPSCR_KIND) SAVED_FPSCR
INTEGER(FP_MODE_KIND) FP_MODE

SAVED_FPSCR = get_fpscr()           ! Saves the current value of
                                     ! the fpscr register.
```

```
CALL set_fpscr_flags(TRP_DIV_BY_ZERO) ! Enables trapping of
! ...                               ! divide-by-zero.
SAVED_FPSCR=set_fpscr(SAVED_FPSCR)  ! Restores fpscr register.
```

Another method is to use the **ieee_set_halting_mode** subroutine in the **ieee_exceptions** module. This portable, elemental subroutine allows you to set the halting (trapping) status for any **FPSCR** exception flags. For example:

```
USE, INTRINSIC :: ieee_exceptions
TYPE(IEEE_STATUS_TYPE) SAVED_FPSCR
CALL ieee_get_status(SAVED_FPSCR) ! Saves the current value of the
! fpscr register

CALL ieee_set_halting_mode(IEEE_DIVIDE_BY_ZERO, .TRUE.) ! Enabled trapping
! ...                                                  ! of divide-by-zero.

CALL IEEE_SET_STATUS(SAVED_FPSCR) ! Restore fpscr register
```

xlf_fp_util Procedures

The **xlf_fp_util** procedures allow you to query and control the floating-point status and control register (fpscr) of the processor directly. These procedures are more efficient than the **fpsets** and **fpgets** subroutines because they are mapped into inlined machine instructions that manipulate the floating-point status and control register directly.

The intrinsic module, **xlf_fp_util**, contains the interfaces and data type definitions for these procedures and the definitions for the named constants that are needed by the procedures. This module enables type checking of these procedures at compile time rather than link time. The following files are supplied for the **xlf_fp_util** module:

File names	File type	Locations
xlf_fp_util.mod	module symbol file (32-bit)	/opt/ibmcmp/xlf/11.1/include
	module symbol file (64-bit)	/opt/ibmcmp/xlf/11.1/include64

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 26. Exception bits to use with `fpsets` and `fpgets`

Array Element to Set to Enable	Array Element to Check if Exception Occurred	Exception Indicated When <code>.TRUE.</code>
n/a	<code>fpstat(fpx)</code>	Floating-point exception summary
n/a	<code>fpstat(fpfex)</code>	Floating-point enabled exception summary
<code>fpstat(fpve)</code>	<code>fpstat(fpvx)</code>	Floating-point invalid operation exception summary
<code>fpstat(fpoe)</code>	<code>fpstat(fpox)</code>	Floating-point overflow exception
<code>fpstat(fpue)</code>	<code>fpstat(fpux)</code>	Floating-point underflow exception
<code>fpstat(fpze)</code>	<code>fpstat(fpzx)</code>	Zero-divide exception
<code>fpstat(fpxe)</code>	<code>fpstat(fpxx)</code>	Inexact exception
<code>fpstat(fpve)</code>	<code>fpstat(fpvxsnan)</code>	Floating-point invalid operation exception (signaling NaN)
<code>fpstat(fpve)</code>	<code>fpstat(fpvxisi)</code>	Floating-point invalid operation exception (INF-INF)
<code>fpstat(fpve)</code>	<code>fpstat(fpvxidi)</code>	Floating-point invalid operation exception (INF/INF)
<code>fpstat(fpve)</code>	<code>fpstat(fpvxzdz)</code>	Floating-point invalid operation exception (0/0)
<code>fpstat(fpve)</code>	<code>fpstat(fpvximz)</code>	Floating-point invalid operation exception (INF*0)
<code>fpstat(fpve)</code>	<code>fpstat(fpvxvc)</code>	Floating-point invalid operation exception (invalid compare)
n/a	<code>fpstat(fpvxsoft)</code>	Floating-point invalid operation exception (software request), PowerPC only
n/a	<code>fpstat(fpvxsqrt)</code>	Floating-point invalid operation exception (invalid square root), PowerPC only
n/a	<code>fpstat(fpvx cvi)</code>	Floating-point invalid operation exception (invalid integer convert), PowerPC only

To explicitly check for specific exceptions at particular points in a program, use `fpgets` and then test whether the elements in `fpstat` have changed. Once an exception has occurred, the corresponding exception bit (second column in the preceding table) is set until it is explicitly reset, except for `fpstat(fpx)`, `fpstat(fpvx)`, and `fpstat(fpfex)`, which are reset only when the specific exception bits are reset.

An advantage of using the `fpgets` and `fpsets` subroutines (as opposed to controlling everything with suboptions of the `-qfltrap` option) includes control over granularity of exception checking. For example, you might only want to test if an exception occurred anywhere in the program when the program ends.

The disadvantages of this approach include the following:

- You have to change your source code.
- These routines differ from what you may be accustomed to on other platforms.

For example, to trap floating-point overflow exceptions but only in a certain section of the program, you would set `fpstat(fpoe)` to `.TRUE.` and call `fpsets`. After the exception occurs, the corresponding exception bit, `fpstat(fpox)`, is `.TRUE.` until the program runs:

```

call fpgets(fpstat)
fpstat(fpox) = .FALSE.
call fpsets(fpstat) ! resetting fpstat(fpox) to .FALSE.

```

Sample programs for exception handling

`/opt/ibmcmp/xlf/11.1/samples/floating_point` contains a number of sample programs to illustrate different aspects of exception handling:

`flttrap_handler.c` and `flttrap_test.f`

A sample exception handler that is written in C and a Fortran program that uses it.

`xl_ieee.F` and `xl_ieee.c`

Exception handlers that are written in Fortran and C that show how to substitute particular values for operations that produce exceptions. Even when you use support code such as this, the implementation of XL Fortran exception handling does not fully support the exception-handling environment that is suggested by the IEEE floating-point standard.

`check_fpscr.f` and `postmortem.f`

Show how to work with the `fpsets` and `fpgets` procedures and the `fpstats` array.

`fhandler.F`

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

`xl_trbk_test.f`

Shows how to use the `xl_trbk` procedure to generate a traceback listing without stopping the program.

The sample programs are strictly for illustrative purposes only.

Causing exceptions for particular variables

To mark a variable as “do not use”, you can encode a special value called a signaling NaN in it. This causes an invalid exception condition any time that variable is used in a calculation.

If you use this technique, use the `nans` suboption of the `-qfloat` option, so that the program properly detects all cases where a signaling NaN is used, and one of the methods already described to generate corresponding `SIGFPE` signals.

Notes:

1. Because a signaling NaN is never generated as the result of a calculation and must be explicitly introduced to your program as a constant or in input data, you should not need to use this technique unless you deliberately use signaling NaN values in it.

Minimizing the performance impact of floating-point exception trapping

If you need to deal with floating-point exception conditions but are concerned that doing so will make your program too slow, here are some techniques that can help minimize the performance impact:

- Consider using only a subset of the `overflow`, `underflow`, `zerodivide`, `invalid`, and `inexact` suboptions with the `-qflttrap` option if you can identify some conditions that will never happen or you do not care about. In particular,

because an **inexact** exception occurs for each rounding error, you probably should not check for it if performance is important.

- Include the **imprecise** suboption with the **-qfltrap** option, so that your compiler command looks similar to this:

```
xlf90 -qfltrap=underflow:enable:imprecise -qsigtrap 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 **-qfltrap** 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 **-qfltrap** (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.

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 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.

Portability of directives

XL Fortran supports many directives available with other Fortran products. This ensures easy portability between products. If your code contains *trigger_constants* other than the defaults in XL Fortran, you can use the **-qdirective** compiler option to specify them. For instance, if you are porting CRAY code contained in a file *xx.f*, you would use the following command to add the CRAY *trigger_constant*:

```
xlf95 xx.f -qdirective=mic\
```

For fixed source form code, in addition to the ! value for the *trigger_head* portion of the directive, XL Fortran also supports the *trigger_head* values **C**, **c**, and *****.

For more information, see the **-qdirective** option in the *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 27. PARALLEL DO Clauses and their XL Fortran synonyms

PARALLEL DO Clause	XL Fortran Synonym
LASTLOCAL	LASTPRIVATE
LOCAL	PRIVATE
MP_SCHEDTYPE and CHUNK	SCHEDULE
SAVELAST	LASTPRIVATE
SHARE	SHARED
NEW	PRIVATE

Table 28. PARALLEL DO scheduling types and their XL Fortran synonyms

Scheduling Type	XL Fortran Synonym
GSS	GUIDED
INTERLEAVE	STATIC(1)
INTERLEAVED	STATIC(1)
INTERLEAVE(n)	STATIC(n)
INTERLEAVED(n)	STATIC(n)
SIMPLE	STATIC

Table 29. PARALLEL SECTIONS clauses and their XL Fortran synonyms

PARALLEL SECTIONS Clause	XL Fortran Synonym
LOCAL	PRIVATE
SHARE	SHARED
NEW	PRIVATE

Common industry extensions that XL Fortran supports

XL Fortran allows many of the same FORTRAN 77 extensions as other popular compilers, including:

Extension	Refer to XL Fortran Language Reference Section(s)
Typeless constants	Typeless Literal Constants
*len length specifiers for types	The Data Types
BYTE data type	BYTE
Long variable names	Names
Lower case	Names
Mixing integers and logicals (with -qintlog option)	Evaluation of Expressions
Character-count Q edit descriptor (with -qqcount option)	Q (Character Count) Editing

Extension	Refer to XL Fortran Language Reference Section(s)
Intrinsics for counting set bits in registers and determining data-object parity	POPCNT, POPPAR
64-bit data types (INTEGER(8) , REAL(8) , COMPLEX(8) , and LOGICAL(8)), including support for default 64-bit types (with -qintsize and -qrealize options)	<i>Integer Real Complex Logical</i>
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.)	POINTER(integer)
Conditional vector merge (CVMGx) intrinsic functions	<i>CVMGx (TSOURCE, FSOURCE, MASK)</i>
Date and time service and utility functions (rtc, irtc, jdate, clock_, timef, and date)	<i>Service and utility procedures</i>
STRUCTURE , UNION , and MAP constructs	<i>Structure Components, Union and Map</i>

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 **-qrealize** 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 237.

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 the **-qlanglvl** and **-qport** options in the *XL Fortran Compiler Reference*.

Appendix. Sample Fortran programs

The following programs are provided as coding examples for XL Fortran. A number of these samples illustrate various aspects of SMP programming that may be new to many users. If you are new to SMP programming, you should examine these samples to gain a better understanding of the SMP coding style. Every attempt has been made to internally document key areas of the source to assist you in this effort.

You can compile and execute the first program to verify that the compiler is installed correctly and your user ID is set up to execute Fortran programs.

Example 1 - XL Fortran source file

```
PROGRAM CALCULATE
!
! Program to calculate the sum of up to n values of x**3
! where negative values are ignored.
!
  IMPLICIT NONE
  INTEGER I,N
  REAL SUM,X,Y
  READ(*,*) N
  SUM=0
  DO I=1,N
    READ(*,*) X
    IF (X.GE.0.0) THEN
      Y=X**3
      SUM=SUM+Y
    END IF
  END DO
  WRITE(*,*) 'This is the sum of the positive cubes:',SUM
END
```

Execution results

Here is what happens when you run the program:

```
$ a.out
5
37
22
-4
19
6
This is the sum of the positive cubes: 68376.00000
```

Example 2 - valid C routine source file

```
/*
 * *****
 * This is a main function that creates threads to execute the Fortran
 * test subroutines.
 * *****
 */
#include <pthread.h>
#include <stdio.h>
#include <errno.h>

extern char *optarg;
extern int optind;

static char *prog_name;

#define MAX_NUM_THREADS 100

void *f_mt_exec(void *);
void f_pre_mt_exec(void);
void f_post_mt_exec(int *);

void
usage(void)
{
    fprintf(stderr, "Usage: %s -t number_of_threads.\n", prog_name);
    exit(-1);
}

main(int argc, char *argv[])
{
    int i, c, rc;
    int num_of_threads, n[MAX_NUM_THREADS];
    char *num_of_threads_p;
    pthread_attr_t attr;
    pthread_t tid[MAX_NUM_THREADS];

    prog_name = argv[0];
    while ((c = getopt(argc, argv, "t")) != EOF)
    {
        switch (c)
        {
            case 't':
                break;

            default:
                usage();
                break;
        }
    }

    argc -= optind;
    argv += optind;
    if (argc < 1)
    {
        usage();
    }

    num_of_threads_p = argv[0];
    if ((num_of_threads = atoi(num_of_threads_p)) == 0)
    {
        fprintf(stderr,
            "%s: Invalid number of threads to be created <%s>\n", prog_name,
                num_of_threads_p);
        exit(1);
    }
}
```

```

}
else if (num_of_threads > MAX_NUM_THREADS)
{
    fprintf(stderr,
            "%s: Cannot create more than 100 threads.\n", prog_name);
    exit(1);
}
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

/* *****
 * Execute the Fortran subroutine that prepares for multi-threaded
 * execution.
 * *****
 */
f_pre_mt_exec();

for (i = 0; i < num_of_threads; i++)
{
    n[i] = i;
    rc = pthread_create(&tid[i], &attr, f_mt_exec, (void *)&n[i]);
    if (rc != 0)
    {
        fprintf(stderr, "Failed to create thread %d.\n", i);

        exit(1);
    }
}
/* The attribute is no longer needed after threads are created. */
pthread_attr_destroy(&attr);
for (i = 0; i < num_of_threads; i++)
{
    rc = pthread_join(tid[i], NULL);
    if (rc != 0)
    {
        fprintf(stderr, "Failed to join thread %d. \n", i);
    }
}
/*
 * Execute the Fortran subroutine that does the check after
 * multi-threaded execution.
 */
f_post_mt_exec(&num_of_threads);

exit(0);
}

```

```

! *****
! This test case tests the writing list-directed to a single external
! file by many threads.
! *****

```

```

subroutine f_pre_mt_exec()
integer array(1000)
common /x/ array

do i = 1, 1000
    array(i) = i
end do

open(10, file="fun10.out", form="formatted", status="replace")
end

subroutine f_post_mt_exec(number_of_threads)
integer array(1000), array1(1000)
common /x/ array

```

```

close(10)
open(10, file="fun10.out", form="formatted")
do j = 1, number_of_threads
  read(10, *) array1

  do i = 1, 1000
    if (array1(i) /= array(i)) then
      print *, "Result is wrong."
      stop
    endif
  end do
end do
close(10, status="delete")
print *, "Normal ending."
end

subroutine f_mt_exec(thread_number)
integer thread_number
integer array(1000)
common /x/ array

write(10, *) array
end

```

Example 3 - valid Fortran SMP source file

```

!*****
!* This example uses a PARALLEL construct and a DO construct *
!* to calculate the value of pi. *
!*****
program compute_pi
integer n, i
real*8 w, x, pi, f, a
f(a) = 4.d0 / (1.d0 + a*a) !! function to integrate

pi = 0.0d0
!$OMP PARALLEL private(x, w, n), shared(pi)
n = 10000 !! number of intervals
w = 1.0d0/n !! calculate the interval size
!$OMP DO reduction(+: pi)
do i = 1, n
  x = w * (i - 0.5d0)
  pi = pi + f(x)
enddo
!$OMP END DO
!$OMP END PARALLEL
print *, "Computed pi = ", pi
end

```

Example 4 - invalid Fortran SMP source file

```

!*****
!* In this example, fort_sub is invoked by multiple threads. *
!* * *
!* This example is not valid because *
!* fort_sub and another_sub both declare /block/ to be *
!* THREADLOCAL. They intend to share the common block, but *
!* they are executed via different threads. *
!* * *
!* To "fix" this problem, one of the following approaches can *
!* be taken: *
!* (1) The code for another_sub should be brought into the loop.*
!* (2) "j" should be passed as an argument to another_sub, and *
!* the declaration for /block/ should be removed from *
!* another_sub. *

```

```

!* (3) The loop should be marked as "do not parallelize" by      *
!*      using the directive "!SMP$ PARALLEL DO IF(.FALSE.)".    *
!*****
subroutine fort_sub()

  common /block/ j
  integer :: j
  !IBM* THREADLOCAL /block/          ! Each thread executing fort_sub
                                      ! obtains its own copy of /block/.

  integer a(10)

  ...
  !IBM* INDEPENDENT
  do index = 1,10
    call another_sub(a(i))
  enddo
  ...

end subroutine fort_sub

subroutine another_sub(aa)          ! Multiple threads are used to
  integer aa                       ! execute another_sub.
  common /block/ j                 ! Each thread obtains a new copy
  integer :: j                     ! of the common block /block/.
  !IBM* THREADLOCAL /block/

  aa = j                           ! The value of "j" is undefined.
end subroutine another_sub

```

Programming examples using the Pthreads library module

```

!*****
!* Example 5 : Create a thread with Round_Robin scheduling policy.*
!* For simplicity, we do not show any codes for error checking,  *
!* which would be necessary in a real program.                   *
!*****
  use, intrinsic::f_pthread
  integer(4) ret_val
  type(f_pthread_attr_t) attr
  type(f_pthread_t) thr

  ret_val = f_pthread_attr_init(attr)
  ret_val = f_pthread_attr_setschedpolicy(attr, SCHED_RR)
  ret_val = f_pthread_attr_setinheritsched(attr, PTHREAD_EXPLICIT_SCHED)
  ret_val = f_pthread_create(thr, attr, FLAG_DEFAULT, ent, integer_arg)
  ret_val = f_pthread_attr_destroy(attr)
  .....

```

Before you can manipulate a pthread attribute object, you need to create and initialize it. The appropriate interfaces must be called to manipulate the attribute objects. A call to `f_pthread_attr_setschedpolicy` sets the scheduling policy attribute to Round_Robin. Note that this does not affect newly created threads that inherit the scheduling property from the creating thread. For these threads, we explicitly call `f_pthread_attr_setinheritsched` to override the default inheritance attribute. The rest of the code is self-explanatory.

```

!*****
!* Example 6 : Thread safety                                     *
!* In this example, we show that thread safety can be achieved  *
!* by using the push-pop cleanup stack for each thread. We     *
!* assume that the thread is in deferred cancellability-enabled *
!* state. This means that any thread-cancel requests will be   *
!* put on hold until a cancellation point is encountered.      *
!* Note that f_pthread_cond_wait provides a                     *
!* cancellation point.                                         *
!*****

```

```

!*****
use, intrinsic::f_thread
integer(4) ret_val
type(f_thread_mutex_t) mutex
type(f_thread_cond_t) cond
pointer(p, byte)
! Initialize mutex and condition variables before using them.
! For global variables this should be done in a module, so that they
! can be used by all threads. If they are local, other threads
! will not see them. Furthermore, they must be managed carefully
! (for example, destroy them before returning, to avoid dangling and
! undefined objects).
mutex = PTHREAD_MUTEX_INITIALIZER
cond = PTHREAD_COND_INITIALIZER

.....
! Doing something

.....

! This thread needs to allocate some memory area used to
! synchronize with other threads. However, when it waits on a
! condition variable, this thread may be canceled by another
! thread. The allocated memory may be lost if no measures are
! taken in advance. This will cause memory leakage.

ret_val = f_thread_mutex_lock(mutex)
p = malloc(%val(4096))

! Check condition. If it is not true, wait for it.
! This should be a loop.

! Since memory has been allocated, cleanup must be registered
! for safety during condition waiting.

ret_val = f_thread_cleanup_push(mycleanup, FLAG_DEFAULT, p)
ret_val = f_thread_cond_wait(cond, mutex)

! If this thread returns from condition waiting, the cleanup
! should be de-registered.

call f_thread_cleanup_pop(0)      ! not execute
ret_val = f_thread_mutex_unlock(mutex)

! This thread will take care of p for the rest of its life.
.....

! mycleanup looks like:

subroutine mycleanup(passed_in)
  pointer(passed_in, byte)
  external free

  call free(%val(passed_in))
end subroutine mycleanup

```

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, 2007. 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

Company, product, or service names identified in the text may be trademarks or service marks of IBM or other companies. Information on the trademarks of International Business Machines Corporation in the United States, other countries, or both is located at <http://www.ibm.com/legal/copytrade.shtml>.

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

Intel is a trademark or registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

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

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.

Index

Special characters

- _OPENMP C preprocessor macro 55
- O0 3
- O2 3
- O3 5
 - trade-offs 5
- O4 6
 - trade-offs 7
- O5 7
 - trade-offs 7
- Q inlining
 - code size 36
- qarch 22
- qcache 6, 24
- qcompact
 - code size 36, 37
- qfloat compiler option 237
 - fltint suboption 237
 - hssngl suboption 237
 - nans suboption 243
 - nomaf suboption 237
 - rsqrt suboption 237
- qflttrap compiler option 238
- qipa 6
 - IPA process 6
- qpdf 13
- qposition compiler option 215
- qsigtrap compiler option 239
- qstrict 5
- qtune 24
- /tmp directory
 - See TMPDIR environment variable
- * length specifiers (FORTRAN 77 extension) 246
- %REF functions 197
- %VAL functions 197

Numerics

- 64-bit data types (FORTRAN 77 extension) 246

A

- advanced optimization 4
- aliasing 31
- arguments
 - passing between languages 192, 193
 - passing by reference or by value 197
- array initialization
 - code size 36
- arrays
 - passing between languages 195
- assembler
 - low-level linkage conventions 200
- ATOMIC 66

B

- BARRIER 68
- basic optimization 2
- bitwise-identical floating-point results 237
- block
 - cyclic scheduling 126
- blocked special files, interaction of XL Fortran I/O with 216
- buffers, flushing 218
- BYTE data type (FORTRAN 77 extension) 246

C

- C language and interlanguage calls 189, 192
- C++ and Fortran in same program 191
- calling by reference or value 197
- calling non-Fortran procedures 189
- CASE construct
 - code size 37
- character data, passing between languages 194
- character special files, interaction of XL Fortran I/O with 216
- character-count edit descriptor (FORTRAN 77 extension) 246
- check_fpscr.f sample file 243
- chunk
 - SCHEDULE directive and 91
- code size 35
 - Q inlining 36
 - qcompact 36, 37
 - array initialization 36
 - blocking 35
 - CASE constructs 37
 - computed GOTOs 37
 - dynamic linking 38
 - high activity areas 37
 - page faults 37
 - static linking 38
 - steps for reduction 36
- common blocks
 - in data scope attribute clauses 111
- Compiler-friendly techniques 41
 - aliasing 42
 - arrays 42
 - choosing appropriate variable sizes 42
 - compiler invocations 41
 - floating-point precision 42
 - PERMUTATION 42
 - pointers 42
 - variables 42
 - WHERE constructs 42
- computed GOTO
 - code size 37

- conditional vector merge intrinsic functions (FORTRAN 77 extension) 246
- control and status register for floating point 241
- controlling optimization transformations 27
- COPYIN clause 112
- core file 239
- CRAY functions (FORTRAN 77 extension)
 - conditional vector merge intrinsics 246
 - date and time service and utility functions 247
- CRAY pointer (FORTRAN 77 extension), XL Fortran equivalent 246
- CRITICAL 69
- CVMGx intrinsic functions (FORTRAN 77 extension) 246

D

- data scope attribute clauses
 - COPYIN clause 112
 - discussion 111
- data types in Fortran, C 193
- date and time functions (FORTRAN 77 extension) 247
- debugging 15
- delays runtime option 59
- Directive Clauses, Global Rules 111
- directives
 - DO SERIAL 74
 - END MASTER 78
 - END ORDERED 79
 - END PARALLEL 81
 - END PARALLEL DO 84
 - END PARALLEL SECTIONS 87
 - END PARALLEL WORKSHARE 90
 - END SECTIONS 94
 - FLUSH 75
 - MASTER 78
 - ORDERED 79
 - PARALLEL 81
 - PARALLEL DO 84
 - PARALLEL SECTIONS 87
 - PARALLEL WORKSHARE 90
 - SCHEDULE 91
 - SECTIONS 94
 - SINGLE / END SINGLE 98
 - THREADLOCAL 101
 - THREADPRIVATE 104
 - WORKSHARE 108
- Directives
 - ATOMIC 66
 - BARRIER 68
 - CRITICAL 69
 - DO (worksharing) 71
 - END CRITICAL 69
 - END DO (worksharing) 71

disk striping
See data striping
 DO (worksharing) 71
 DO SERIAL compiler directive 74
 double-precision values 230, 232
 dynamic linking
 code size 38

E

enable suboption of -qfltrap 240
 enabling MASS 9
 enabling VMX 9
 END CRITICAL 69
 END DO (worksharing) 71
 END MASTER compiler directive 78
 END ORDERED compiler directive 79
 END PARALLEL compiler directive 81
 END PARALLEL DO compiler
 directive 84
 END PARALLEL SECTIONS compiler
 directive 87
 END PARALLEL WORKSHARE compiler
 directive 90
 END SECTIONS compiler directive 94
 environment variables
 OpenMP
 OMP_DYNAMIC 61
 OMP_NESTED 62
 OMP_NUM_THREADS 62
 OMP_SCHEDULE 63
 runtime
 XLSMPOPTS 56
 example programs
 See sample programs
 exception handling 231
 for floating point 238
 installing an exception handler 239
 execution environment routines
 OpenMP 129
 explicit interfaces 199
 extended-precision values 233
 extensions to FORTRAN 77, list of
 common ones 246

F

f_maketime function 147
 f_pthread 144
 f_pthread_attr_destroy function 147
 f_pthread_attr_getdetachstate
 function 147
 f_pthread_attr_getguardsize
 function 148
 f_pthread_attr_getinheritsched
 function 149
 f_pthread_attr_getschedparam
 function 149
 f_pthread_attr_getschedpolicy
 function 150
 f_pthread_attr_getscope function 150
 f_pthread_attr_getstack function 151
 f_pthread_attr_init function 151
 f_pthread_attr_setdetachstate
 function 152
 f_pthread_attr_setguardsize 152

f_pthread_attr_setinheritsched
 function 153
 f_pthread_attr_setschedparam
 function 153
 f_pthread_attr_setschedpolicy
 function 154
 f_pthread_attr_setscope function 154
 f_pthread_attr_setstack function 155
 f_pthread_attr_t function 156
 f_pthread_cancel function 156
 f_pthread_cleanup_pop function 156
 f_pthread_cleanup_push function 157
 f_pthread_cond_broadcast function 158
 f_pthread_cond_destroy function 158
 f_pthread_cond_init function 159
 f_pthread_cond_signal function 159
 f_pthread_cond_t function 160
 f_pthread_cond_timedwait function 160
 f_pthread_cond_wait function 161
 f_pthread_condattr_destroy function 161
 f_pthread_condattr_getpshared
 function 162
 f_pthread_condattr_init function 162
 f_pthread_condattr_setpshared
 function 163
 f_pthread_condattr_t function 163
 f_pthread_create function 164
 f_pthread_detach function 165
 f_pthread_equal function 165
 f_pthread_exit function 166
 f_pthread_getconcurrency function 166
 f_pthread_getschedparam function 167
 f_pthread_getspecific function 167
 f_pthread_join function 168
 f_pthread_key_create function 168
 f_pthread_key_delete function 169
 f_pthread_key_t 170
 f_pthread_kill function 170
 f_pthread_mutex_destroy function 170
 f_pthread_mutex_init function 171
 f_pthread_mutex_lock function 171
 f_pthread_mutex_t 172
 f_pthread_mutex_trylock function 172
 f_pthread_mutex_unlock function 172
 f_pthread_mutexattr_destroy
 function 173
 f_pthread_mutexattr_getpshared
 function 173
 f_pthread_mutexattr_gettype
 function 174
 f_pthread_mutexattr_init function 175
 f_pthread_mutexattr_setpshared
 function 175
 f_pthread_mutexattr_settype
 function 176
 f_pthread_mutexattr_t 177
 f_pthread_once function 177
 f_pthread_once_t 177
 f_pthread_rwlock_destroy function 177
 f_pthread_rwlock_init function 178
 f_pthread_rwlock_rdlock function 178
 f_pthread_rwlock_t function 179
 f_pthread_rwlock_tryrdlock
 function 179
 f_pthread_rwlock_trywrlock
 function 180
 f_pthread_rwlock_unlock function 180

f_pthread_rwlock_wlock function 181
 f_pthread_rwlockattr_destroy
 function 181
 f_pthread_rwlockattr_getpshared
 function 182
 f_pthread_rwlockattr_init function 182
 f_pthread_rwlockattr_setpshared
 function 183
 f_pthread_rwlockattr_t function 183
 f_pthread_self function 184
 f_pthread_setcancelstate function 184
 f_pthread_setcanceltype function 185
 f_pthread_setconcurrency function 185
 f_pthread_setschedparam function 186
 f_pthread_setspecific function 186
 f_pthread_t function 187
 f_pthread_testcancel function 187
 f_sched_param function 187
 f_sched_yield function 188
 f_timespec function 188
 fexcp.h include file 239
 fhandler.F sample file 243
 file positioning 215
 files
 I/O formats 213
 names 214
 permissions 217
 floating-point
 exceptions 238
 processing 229
 optimizing 237
 floating-point optimization 25
 floating-point status and control
 register 241
 FLTRAP @PROCESS directive 238
 flttrap_handler.c and flttrap_test.f sample
 files 243
 FLUSH compiler directive 75
 flushing I/O buffers 218
 formats, file 213
 fort.* default file names 214, 219
 FORTRAN 77 extensions, list of common
 ones 246
 fpdt.h and fpdc.h include files 234
 fpgets and fpsets service and utility
 subroutines 241
 fpscr register 241
 fpstat array 241
 functions
 linkage convention for calls 209
 return values 200

G

get_round_mode procedure 234
 GETENV intrinsic procedure 214

H

high-order transformation 8
 HOT 8

I

I/O
 See input/output

IEEE arithmetic 229
 implicitly connected files 214
 include files `fpdt.h` and `fpdc.h` 234
 infinity values 230
 initial file position 215
 Inlining 32
 input/output 229
 from two languages in the same program 190
 redirection 216
 XL Fortran implementation details 213
 instruction group support by architecture 22
 integer POINTER (FORTRAN 77 extension) 246
 interlanguage calls 189, 197
 arrays 195
 C++ 191
 character types 194
 corresponding data types 193
 input and output 190
 low-level linkage conventions 200
 pointers 196
 IPA 10
 levels 11
 suboptions 11
 with C and C++ 12

L

libmass 50
 libmass library 43
 libmassv library 45
 library
 MASS 43
 scalar 43
 vector 45
 links, interaction of XL Fortran I/O with 216
 Linux Pthreads Library 144
 lock routines
 OpenMP 129
 long variable names (FORTRAN 77 extension) 246
 long vectorization 9
 loop optimization 8
 lower case (FORTRAN 77 extension) 246

M

macro, `_OPENMP` C preprocessor 55
 main, restriction on use as a Fortran name 189
 MASS libraries 43
 scalar functions 43
 vector functions 45
 MASTER compiler directive 78
 migrating
 from other systems 245
 minus infinity, representation of 230
 mixing integers and logicals (FORTRAN 77 extension) 246
 module procedures, external names corresponding to 189

N

naming conventions for external names 189
 NaN values
 and infinities 230
 negative infinity, representation of 230
 null-terminated strings, passing to C functions 194

O

omp_destory_nest_lock OpenMP nestable lock routine 130
 omp_destroy_lock OpenMP lock routine 130
 OMP_DYNAMIC environment variable 61
 omp_get_dynamic execution environment routine 131
 omp_get_max_threads execution environment routine 131
 omp_get_nestded execution environment routine 132
 omp_get_num_procs execution environment routine 132
 omp_get_num_threads execution environment routine 132
 omp_get_thread_num execution environment routine 133
 omp_get_wtick OpenMP timing routine 134
 omp_get_wtime OpenMP timing routine 135
 omp_in_parallel execution environment routine 135
 omp_init_lock lock routine 136
 omp_init_nest_lock OpenMP nestable lock routine 137
 OMP_NESTED environment variable 62
 OMP_NUM_THREADS environment variable 62
 OMP_SCHEDULE environment variable 63
 omp_set_dynamic execution environment routine 138
 omp_set_lock lock routine 138
 omp_set_nest_lock nestable lock routine 140
 omp_set_nestded execution environment routine 139
 omp_set_num_threads execution environment routine 141
 omp_test_lock lock routine 141
 omp_test_nest_lock lock routine 142
 omp_unset_lock lock routine 142
 omp_unset_nest_lock lock routine 143
 OpenMP
 execution environment routines description 129
 omp_get_dynamic 131
 omp_get_max_threads 131
 omp_get_nestded 132
 omp_get_num_procs 132
 omp_get_num_threads 132
 omp_get_thread_num 133
 omp_in_parallel 135

OpenMP (continued)
 execution environment routines (continued)
 omp_set_dynamic 138
 omp_set_nestded 139
 omp_set_num_threads 141
 lock routines
 description 129
 omp_destroy_lock 130
 omp_init_lock 136
 omp_set_lock 138
 omp_test_lock 141
 omp_test_nest_lock 142
 omp_unset_lock 142
 omp_unset_nest_lock 143
 nestable lock routines
 omp_destroy_nest_lock 130
 omp_init_nest_lock 137
 omp_set_nest_lock 140
 timing routines
 omp_get_wtick 134
 omp_get_wtime 135
 OpenMP environment variables 61
 optimization
 -O0 3
 -O2 3
 -O3 5
 -O4 6
 -O5 7
 advanced 4
 and tuning 21
 basic 2
 debugging 15
 floating-point 25, 27
 for floating-point arithmetic 237
 loops 25, 27
 math functions 43
 options to avoid 29
 PDF 13
 profile directed feedback 13
 optimization trade-offs
 -O3 5
 -O4 7
 -O5 7
 option driven tuning 25
 OPTIONAL attribute 200
 options for targeting your architecture 21
 ORDERED compiler directive 79

P

PARALLEL compiler directive discussion 81
 PARALLEL DO compiler directive discussion 84
 SCHEDULE clause 84
 parallel execution options 58
 PARALLEL SECTIONS compiler directive discussion 87
 PARALLEL WORKSHARE compiler directive discussion 90
 parameters
 See arguments
 parthds runtime option 58
 partheshold runtime option 60

- Pascal language and interlanguage calls 189
- PDF 13
 - walkthrough 13
- performance analysis 29
- performance of floating-point arithmetic 237
- performance tuning options 59
- permissions of files 217
- pipes, interaction of XL Fortran I/O with 216
- plus infinity, representation of 230
- pointers (integer POINTER) (FORTRAN 77 extension) 246
- portability 245
- porting to XL Fortran 245
- POSITION @PROCESS directive 215
- position of a file after an OPEN statement 215
- positive infinity, representation of 230
- postmortem.f sample file 243
- preconnected files 214
- profilefreq runtime option 60
- providing your application characteristics 25
- pseudo-devices, interaction of XL Fortran I/O with 216
- pthread library module 253
- Pthreads Library Module
 - descriptions of functions in 144
 - f_maketime function 147
 - f_pthread_attr_destroy function 147
 - f_pthread_attr_getdetachstate function 147
 - f_pthread_attr_getguardsize function 148
 - f_pthread_attr_getinheritsched function 149
 - f_pthread_attr_getschedparam function 149
 - f_pthread_attr_getschedpolicy function 150
 - f_pthread_attr_getscope function 150
 - f_pthread_attr_getstack 151
 - f_pthread_attr_init function 151
 - f_pthread_attr_setdetachstate function 152
 - f_pthread_attr_setguardsize function 152
 - f_pthread_attr_setinheritsched function 153
 - f_pthread_attr_setschedparam function 153
 - f_pthread_attr_setschedpolicy function 154
 - f_pthread_attr_setscope function 154
 - f_pthread_attr_setstack function 155
 - f_pthread_attr_t function 156
 - f_pthread_cancel function 156
 - f_pthread_cleanup_pop function 156
 - f_pthread_cleanup_push function 157
 - f_pthread_cond_broadcast function 158
 - f_pthread_cond_destroy function 158
 - f_pthread_cond_init function 159
 - f_pthread_cond_signal function 159

- Pthreads Library Module (*continued*)
 - f_pthread_cond_t function 160
 - f_pthread_cond_timedwait function 160
 - f_pthread_cond_wait function 161
 - f_pthread_condattr_destroy function 161
 - f_pthread_condattr_getpshared function 162
 - f_pthread_condattr_init function 162
 - f_pthread_condattr_setpshared function 163
 - f_pthread_condattr_t function 163
 - f_pthread_create function 164
 - f_pthread_detach function 165
 - f_pthread_equal function 165
 - f_pthread_exit function 166
 - f_pthread_getconcurrency function 166
 - f_pthread_getschedparam function 167
 - f_pthread_getspecific function 167
 - f_pthread_join function 168
 - f_pthread_key_create function 168
 - f_pthread_key_delete function 169
 - f_pthread_key_t 170
 - f_pthread_kill function 170
 - f_pthread_mutex_destroy function 170
 - f_pthread_mutex_init function 171
 - f_pthread_mutex_lock function 171
 - f_pthread_mutex_t 172
 - f_pthread_mutex_trylock function 172
 - f_pthread_mutex_unlock function 172
 - f_pthread_mutexattr_destroy function 173
 - f_pthread_mutexattr_getpshared function 173
 - f_pthread_mutexattr_gettype function 174
 - f_pthread_mutexattr_init function 175
 - f_pthread_mutexattr_setpshared function 175
 - f_pthread_mutexattr_settype function 176
 - f_pthread_mutexattr_t 177
 - f_pthread_once function 177
 - f_pthread_once_t 177
 - f_pthread_rwlock_destroy function 177
 - f_pthread_rwlock_init function 178
 - f_pthread_rwlock_rdlock function 178
 - f_pthread_rwlock_t function 179
 - f_pthread_rwlock_tryrdlock function 179
 - f_pthread_rwlock_trywrlock function 180
 - f_pthread_rwlock_unlock function 180
 - f_pthread_rwlock_wrlock function 181
 - f_pthread_rwlockattr_destroy function 181

- Pthreads Library Module (*continued*)
 - f_pthread_rwlockattr_getpshared function 182
 - f_pthread_rwlockattr_init function 182
 - f_pthread_rwlockattr_setpshared function 183
 - f_pthread_rwlockattr_t function 183
 - f_pthread_self function 184
 - f_pthread_setcancelstate function 184
 - f_pthread_setcanceltype function 185
 - f_pthread_setschedparam function 186
 - f_pthread_setconcurrency function 185
 - f_pthread_setspecific function 186
 - f_pthread_t function 187
 - f_pthread_testcancel function 187
 - f_sched_param function 187
 - f_sched_yield function 188
 - f_timespec function 188
- Pthreads Library, Linux 144

Q

- Q (character-count) edit descriptor (FORTRAN 77 extension) 246
- quiet NaN 230

R

- real arithmetic 229
- REAL(16) values 233
- REAL(4) and REAL(8) values 230, 232
- record lengths 217
- redirecting input/output 216
- reference, passing arguments by 197
- rounding 234
 - rounding errors 236
 - rounding mode 234, 236

S

- sample programs 249
 - calling C functions from Fortran 194
 - floating-point exception handling 243
 - for SMP 250
- scalar MASS library 43
- SCHEDULE clause, of PARALLEL DO directive 84
- SCHEDULE compiler directive 91
 - discussion 91
- schedule runtime option 56
- scheduling, block cyclic 126
- scope
 - data scope attribute clauses 111
- scratch file directory
 - See TMPDIR environment variable
- scratch_vars runtime option 219
- SECTIONS compiler directive
 - discussion 94
- seqthreshold runtime option 60
- short vectorization 9
- SIGFPE signal 238, 239

- signal handling
 - for floating point 238
 - installing an exception handler 239
- signaling NaN 230, 243
- SIGTRAP signal 238, 239
- SINGLE / END SINGLE compiler directive 98
- single-precision values 230, 232
- SMP
 - sample programs 250
- special files, interaction of XL Fortran I/O with 216
- spins runtime option 59
- stack 202
- stack runtime option 58
- standard error, input, and output streams 214
- star length specifiers 246
- static linking
 - code size 38
- status and control register for floating point 241
- stderr, stdin, and stdout streams 214
- strings, passing to C functions 194
- subprograms in other languages, calling 189, 192
- Sun pointer (FORTRAN 77 extension), XL Fortran equivalent 246
- symbolic links, interaction of XL Fortran I/O with 216

T

- tctl command 217
- thread visible variables 76
- thread-safing
 - pthreads library module 144
- THREADLOCAL compiler directive 101
- THREADPRIVATE compiler directive 104
- time and date functions (FORTRAN 77 extension) 247
- Trace/BPT trap 239
- traceback listing 240
- tuning 21
 - qarch 22
 - qcache 24
 - qtune 24
 - controlling optimization transformations 27
 - instruction group support by architecture 22
 - option driven tuning 25
 - options to avoid 29
 - performance analysis 29
 - providing your application characteristics 25
- typeless constants (FORTRAN 77 extension) 246

U

- unit_vars runtime option 219
- usrthds runtime option 58

V

- VALUE attribute 197
- value, passing arguments by 197
- vector MASS library 45
- vectorization
 - long 9
 - short 9

W

- work-sharing constructs
 - SECTIONS / END SECTIONS compiler directives 94
 - SINGLE / END SINGLE compiler directives 98
- WORKSHARE compiler directive 108

X

- xl_ieee exception handler 240
- xl_ieee.F and xl_ieee.c sample files 243
- xl_sigdump exception handler 240
- xl_trbk exception handler 240
- xl_trbk_test.f sample file 243
- xl_trce exception handler 240
- xl_trcedump exception handler 240
- XLFSRATCH_unit environment variable 219
- XLFUNIT_unit environment variable 219
- XLSMPOPTS environment variable 56

Y

- yields runtime option 59



Program Number: 5724-S74

SC23-5898-00

