Intel® Fortran Compiler
User's Guide

Document No. FL-700-05

Disclaimer

Information in this document is provided in connection with Intel products. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. Intel products are not intended for use in medical, life saving, or life sustaining applications.

This *Intel® Fortran Compiler User's Guide* as well as the software described in it is furnished under license and may only be used or copied in accordance with the terms of the license. The information in this document is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Intel Corporation. Intel Corporation assumes no responsibility or liability for any errors or inaccuracies that may appear in this document or any software that may be provided in association with this document.

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

Celeron, Dialogic, i386, i486, iCOMP, Intel, Intel logo, Intel386, Intel486, Intel740, IntelDX2, IntelDX4, IntelSX2, Intel Inside, Intel Inside logo, Intel NetBurst, Intel NetStructure, Intel Xeon, Intel XScale, Itanium, MMX, MMX logo, Pentium, Pentium II Xeon, Pentium III Xeon, and VTune are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

* Other names and brands may be claimed as the property of others.

Welcome to Intel® Fortran Compiler

The Intel® Fortran Compiler version 7.0 compiles code targeted for the IA-32 Intel® architecture and Intel® Itanium® architecture. The Intel Fortran Compiler has a variety of options that enable you to use the compiler features for higher performance of your application.

In addition to the Getting Started with the Intel® Fortran Compiler section included with this document, for installing and more details on getting started, see *Intel® Fortran Compiler Installing and Getting Started* document.

Major Components of the Intel® Fortran Compiler Product

Intel® Fortran Compiler product includes the following components for the development environment:

- Intel® Fortran Compiler for 32-bit Applications
- Intel® Fortran Itanium® Compiler for Itanium-based Applications
- Intel Debugger (IDB)

The Intel Fortran Compiler for Itanium-based applications includes Intel® Itanium® Assembler and Intel Itanium® Linker. This documentation assumes that you are familiar with the Fortran programming language and with the Intel® processor architecture. You should also be familiar with the host computer's operating system.

What's New in This Release

This document combines information about Intel® Fortran Compiler for IA-32-based applications and Itanium®-based applications. IA-32-based applications correspond to the applications run on any processor of the Intel® Pentium® processor family generations, including the Xeon(TM) processor. Itanium-based applications correspond to the applications run on the Intel® Itanium® and Itanium 2 processors.

The following variations of the compiler are provided for you to use according to your host system's processor architecture and targeted architectures.

- Intel® Fortran Compiler for 32-bit Applications is designed for IA-32 systems, and its command is `ifc`. The IA-32 compilations run on any IA-32 Intel processor and produce applications that run on IA-32 systems. This compiler can be optimized specifically for one or more Intel IA-32 processors, from Intel® Pentium® to Pentium 4 to Celeron(TM) and Xeon(TM) processors.
- Intel® Fortran Itanium® Compiler for Itanium®-based Applications (native compiler) is

designed for Itanium architecture systems, and its command is `efc`. This compiler runs on Itanium-based systems and produces Itanium-based applications. Itanium-based compilations can only operate on Itanium-based systems.

Improvements and New Features

- New Intel® Itanium® and Itanium 2 processors support with `-tpp1` and `-tpp2` options
- New OpenMP* option, `-openmp_stubs`
- Support of `.mod` files for parallel invocations and the `-module` option
- Extended optimization directives

The Intel Fortran Compiler has a variety of options that enable you to use the compiler features for higher performance of your application. For new options in this release, see New Compiler Options.

Note
Please refer to the Release Notes for the most current information about features implemented in this release.

Hyper-Threading Technology Support

Both auto-parallelization and OpenMP features support Hyper-Threading Technology. Hyper-Threading Technology enables the operation of multiple logical processors to share execution resources in each physical processor package. It increases system throughput when executing multithreaded applications or when multitasked workloads are running concurrently.

OpenMP* Support

The Intel® Fortran Compiler supports OpenMP API version 2.0 and performs code transformation for shared memory parallel programming. The OpenMP support is accomplished with the `-openmp` option. In addition, the functionality of the OpenMP has been reinforced with new option, `-openmp_stubs`.

Optimizing for Intel® Itanium® 2 Processor Family

New options `-tpp1` and `-tpp2` provide specific support for Intel® Itanium® and Itanium 2 processors.

Support of Parallel Invocations

The programs in which modules are defined support valuable compilation mechanisms, such as parallel invocations with make file for Inter-procedural optimizations of multiple files and of the whole program. In addition, the programs that require modules located in multiple directories, can be compiled using the `-I`*`dir`* option to locate the `.mod` files (modules) that should be included in the program. The new `-module` option specifies the directory to rout the module files.

Extended Optimization Directives

In addition to the compiler options, Intel Fortran Compiler supports Intel-extended language directives perform various tasks during compilation to enhance optimization of application code. A few directives for software pipelining, loop unrolling and prefetching have been added.

Features and Benefits

The Intel® Fortran Compiler enables your software to perform the best on Intel architecture-based computers. Using new compiler optimizations, such as the whole-program optimization and profile-guided optimization, prefetch instruction and support for Streaming SIMD Extensions (SSE) and Streaming SIMD Extensions 2 (SSE2), the Intel Fortran Compiler provides high performance.

Feature	Benefit
High Performance	Achieve a significant performance gain by using optimizations
Support for Streaming SIMD Extensions	Advantage of new Intel microarchitecture
Automatic vectorizer	Advantage of parallelism in your code achieved automatically
Parallelization	Automatic generation of multithreaded code for loops. Shared memory parallel programming with OpenMP*.
Floating-point optimizations	Improved floating-point performance
Data prefetching	Improved performance due to the accelerated data delivery
Interprocedural optimizations	Larger application source files perform better
Whole program optimization	Improved performance between modules in larger applications
Profile-guided optimization	Improved performance based on profiling the frequently used procedure
Processor dispatch	Taking advantage of the latest Intel architecture features while maintaining object code compatibility with previous generations of Intel® Pentium® Processors.

Product Web Site and Support

For the latest information about Intel Fortran Compiler, visit the Intel® Fortran Compiler home page where you can find:

- Fortran compiler performance-related information
- Marketing information
- Internet-based support and resources
- Intel Architecture Performance Training Center

For general information on Intel® software development products, visit http://www.intel.com/software/products/index.htm.

For specific details on the Itanium® architecture, visit the web site at http://developer.intel.com/design/itanium/index.htm?iid=search+Itanium&.

System Requirements

The Intel® Fortran Compiler can be run on personal computers that are based on Intel® architecture processors. To compile programs with this compiler, you need to meet the processor and operating system requirements.

Minimum Hardware Requirements

IA-32 Compiler

- A system based on a Pentium®, Intel® Xeon(TM) processor or subsequent IA-32 processor.
- 128 MB RAM
- 100 MB disk space

Recommended: A system with Pentium 4 or Xeon processor and 256 MB of RAM.

Itanium® Compiler

- Itanium-processor-based system. The Itanium®-based systems are shipped with all of the hardware necessary to support this Itanium® compiler.
- 512 MB RAM (1GB RAM recommended)
- 100 MB disk space

Operating System Requirements

IA-32 architecture:
For the current Linux* versions of kernel and glibc supported, please refer to the product Release Notes.

Itanium® architecture:
To run Itanium®-based applications, you must have an Intel® Itanium® architecture system running the Itanium®-based operating system. Itanium®-based systems are shipped with all of the hardware necessary to support this product. For the current Linux versions of kernel and glibc supported, please refer to the product Release Notes.

It is the responsibility of application developers to ensure that the operating system and processor on which the application is to run support the machine instructions contained in the application.

For use/call-sequence of the libraries, see the library documentation provided in your operating system. For GNU libraries for Fortran, refer to http://www.gnu.org/directory/gcc.html in case they are not installed with your operating system.

Browser

For both architectures, the browser Netscape*, version 4.74 or higher is required.

FLEXlm* Electronic Licensing

The Intel® Fortran Compiler uses the GlobeTrotter* FLEXlm* licensing technology. The compiler requires valid license file in the `licenses` directory in the installation path. The default directory is `/opt/intel/licenses` and the license files have a file extension of `.lic`.

*Using the Intel® License Manager for FLEXlm** describes how to install and use the Intel® License Manager for FLEXlm to configure a license server for systems using counted licenses.

How to Use This Document

This User's Guide explains how you can use the Intel® Fortran Compiler. It provides information on how to get started with the Intel Fortran Compiler, how this compiler operates and what capabilities it offers for high performance. You will learn how to use the standard and advanced compiler optimizations to gain maximum performance of your application.

This documentation assumes that you are familiar with the Fortran Standard programming language and with the Intel® processor architecture. You should also be familiar with the host computer's operating system.

Note:
This document explains how information and instructions apply differently to each targeted architecture. If there is no specific indication to either architecture, the description is applicable for both architectures.

Notation Conventions

This documentation uses the following conventions:

`This type style`	An element of syntax, a reserved word, a keyword, a file name, or a code example. The text appears in lowercase unless uppercase is required.
`This type style`	Indicates the exact characters you type as input.
`This type style`	Command line arguments and option arguments you enter.
`This type style`	Indicates an argument on a command line or an option's argument in the text.
`[options]`	Indicates that the items enclosed in brackets are optional.
`{value|value}`	A value separated by a vertical bar (|) indicates a version of an option.
`...` (ellipses)	Ellipses in the code examples indicate that part of the code is not shown.
`This type style`	Indicates an Intel Fortran Language extension code example.
This type style	Indicates an Intel Fortran Language extension discussion. Throughout the manual, extensions to the ANSI standard Fortran language appear in this color to help you easily identify when your code uses a non-standard language extension.
This type style	Hypertext

Related Publications

The following documents provide additional information relevant to the Intel Fortran Compiler:

- *Fortran 95 Handbook*, Jeanne C. Adams, Walter S. Brainerd, Jeanne T. Martin, Brian T. Smith, and Jerrold L. Wagener. The MIT Press, 1997. Provides a comprehensive guide to the standard version of the Fortran 95 Language.

- *Fortran 90/95 Explained*, Michael Metcalf and John Reid. Oxford University Press, 1996. Provides a concise description of the Fortran 95 language.

Information about the target architecture is available from Intel and from most technical bookstores. Most Intel documents are available from the Intel Corporation web site at www.intel.com. Some helpful titles are:

- *Intel® Fortran Libraries Reference*, doc. number 687929
- *Intel® Fortran Programmer's Reference*, doc. number 687928
- *Using the Intel® License Manager for FLEXlm**
- VTune(TM) Performance Analyzer online help
- *Intel Architecture Software Developer's Manual*
- Vol. 1: Basic Architecture, Intel Corporation, doc. number 243190
- Vol. 2: Instruction Set Reference Manual, Intel Corporation, doc. number 243191
- Vol. 3: System Programming, Intel Corporation, doc. number 243192
- *Intel® Itanium® Architecture Application Developer's Architecture Guide*
- *Intel® Itanium® Architecture Software Developer's Manual*
- Vol. 1: Application Architecture, Intel Corporation, doc. number 245317
- Vol. 2: System Architecture, Intel Corporation, doc. number 245318
- Vol. 3: Instruction Set Reference, Intel Corporation, doc. number 245319
- Vol. 4: Itanium Processor Programmer's Guide, Intel Corporation, doc. number 245319
- *Intel® Itanium® Architecture Software Conventions & Runtime Architecture Guide*
- *Intel® Itanium® Architecture Assembly Language Reference Guide*
- *Intel® Itanium® Assembler User's Guide*
- *Pentium® Processor Family Developer's Manual*
- *Intel® Processor Identification with the CPUID Instruction*, Intel Corporation, doc. number 241618

For developer's manuals on Intel processors, refer to the Intel's Literature Center.

Publications on Compiler Optimizations

The following sources are useful in helping you understand basic optimization and vectorization terminology and technology:

- *Intel® Architecture Optimization Reference Manual*

- *Dependence Analysis*, Utpal Banerjee (A Book Series on Loop Transformations for Restructuring Compilers). Kluwer Academic Publishers. 1997.
- *The Structure of Computers and Computation*: Volume I, David J. Kuck. John Wiley and Sons, New York, 1978.
- *Loop Transformations for Restructuring Compilers: The Foundations*, Utpal Banerjee (A Book Series on Loop Transformations for Restructuring Compilers). Kluwer Academic Publishers. 1993.
- *Loop Parallelization*, Utpal Banerjee (A Book Series on Loop Transformations for Restructuring Compilers). Kluwer Academic Publishers. 1994.
- *High Performance Compilers for Parallel Computers*, Michael J. Wolfe. Addison-Wesley, Redwood City. 1996.
- *Supercompilers for Parallel and Vector Computers*, H. Zima. ACM Press, New York, 1990.
- *Efficient Exploitation of Parallelism on Pentium® III and Pentium® 4 Processor-Based Systems*, Aart Bik, Milind Girkar, Paul Grey, and Xinmin Tian.

Options Quick Reference Guides

This section provides three sets of tables comprising Intel® Fortran Compiler Options Quick Reference Guides:

- Alphabetical Listing, alphabetic tabular reference of all compiler and compilation as well as linker and linking control, and all other options implemented by the Intel Fortran Compiler available for both IA-32 and Intel® Itanium® compilers as well as those available exclusively for each architecture.
- Summary tables for IA-32 and Itanium compiler features with the options that enable them
- Compiler Options for Windows* and Linux* Cross-reference

Conventions used in the Options Quick Guide Tables

`[-]`	indicates that option is ON by default, and if option includes "-", the option is disabled; for example, `-cerrs-` disables printing errors in a terse format.
`[n]`	indicates that the value in [] can be omitted or have various values; for example, in `-unroll[n]` option, n can be omitted or have different values starting from 0.
Values in {} with vertical bars	are used for option's version; for example, option `-i{2|4|8}` has these versions: `-i2`, `-i4`, `-i8`.
`{n}`	indicates that option must include one of the fixed values for n; for example, in option `-Zp{n}`, n can be equal to 1, 2, 4, 8, 16.
Words in *`this style`* following an `option`	indicate option's required argument(s). Arguments are separated by comma if more than one are required. For example, the option `-Qoption,`*`tool,opts`* looks in the command line like this: *prompt*>**`ifc -Qoption,`*`link,-w myprog.f`*

New Compiler Options

The following table lists new options in this release. See Conventions Used in the Options Quick Guide Tables.

- Options specific to the Itanium® architecture (Itanium®-based systems only)

All other options are available for both IA-32 and Itanium architectures.

Option	Description	Default
`-dynamic-linker(`*`file`*`)`	Specifies in *`file`* a dynamic linker of choice, rather than default.	OFF
`-module[path]` `-nomodule`	Specifies the directory where the module files (extension `.mod`) are placed. Omitting this option or specifying `-nomodule` results in placing the `.mod` files in the directory where the source files are being compiled. More...	`-nomodule`
`-Ob{0|1|2}`	Controls the compiler's inline expansion. The amount of inline expansion performed varies as follows: `-Ob0`: disable inlining `-Ob1`: disables inlining unless `-ip` or `-Ob2` is specified. Enables inlining of functions. `-Ob2`: Enables inlining of any function. However, the compiler decides which functions are inlined. This option enables interprocedural optimizations and has the same effect as specifying the `-ip` option.	`-Ob1`
`-openmp_stubs`	Enables to compile OpenMP programs in sequential mode. The OpenMP directives are ignored and a stub OpenMP library is linked (sequentially).	OFF
`-safe_cray_ptr`	Specifies that Cray pointers do not alias with other variables. More...	OFF
`-list`	Prints a source listing on `stdout`. More...	OFF
`-list -show`*`include`*	Prints a source listing to `stdout` with contents of `INCLUDE` files. More...	OFF

`-tpp1` Itanium®-based systems	Targets optimization to the Intel® Itanium® processor for best performance. More...	OFF
`-tpp2` Itanium-based systems	Targets optimization to the Intel® Itanium® 2 processor for best performance. Generated code is compatible with the Itanium processor. More...	ON

Compiler Options Quick Reference Alphabetical

The following table describes options that you can use for compilations you target to either IA-32- or Itanium®-based applications or both. See Conventions Used in the Options Quick Guide Tables.

- Options specific to IA-32 architecture (IA-32 only)
- Options specific to the Itanium® architecture (Itanium-based systems only)

All other options are available for both IA-32 and Itanium architectures.

Option	Description	Default
`-0f_check` IA-32 compiler	Enables a software patch for Pentium® processor `0f` erratum. More...	OFF
`-1`	Executes any `DO` loop at least once. Same as `-onetrip`. More...	OFF
`-72, -80, -132`	Specifies 72, 80 or 132 column lines for fixed form source only. The compiler might issue a warning for non-numeric text beyond 72 for the `-72` option. More...	`-72`
`-align`	Analyzes and reorders memory layout for variables and arrays. More... To disable, use the `-noalign` option (default is OFF)	ON

`-ansi_alias[-]`	Enables (default) or disables assumption of the programs ANSI conformance. More...	ON
`-auto`	Makes all local variables `AUTOMATIC`. More...	OFF
`-autodouble`	Sets the default size of real numbers to 8 bytes; same as `-r8`. More...	OFF
`-auto_scalar`	Makes scalar local variables `AUTOMATIC`. More...	ON
`-ax{i|M|K|W}` IA-32 compiler	Generates processor-specific code corresponding to one of codes: `i`, `M`, `K`, and `W` while also generating generic IA-32 code. Compiler generates multiple versions of some routines, and chooses the best version for the host processor at runtime indicated by processor-specific codes `i` (Pentium® Pro), `M` (Pentium with MMX(TM) technology), `K` (Pentium III), and `W` (Pentium 4 and Xeon(TM)). More...	OFF
`-Bdynamic`	Used with `-l`*name* (see in this table), enables dynamic linking of libraries at run time. Compared to static linking, results in smaller executables.	OFF
`-Bstatic`	Enables linking a user's library statically.	OFF
`-c`	Stops the compilation process after an object file (`.o`) has been generated. More...	OFF

`-C90`	Links with an alternative I/O library (`libCEPCF90.a`) that supports mixed input and output with C on the standard streams. More...	OFF
`-C` IA-32 compiler	Equivalent to: (`-CA`, `-CB`, `-CS`, `-CU`, `-CV`) extensive runtime diagnostics options. More...	OFF
`-CA` IA-32 compiler	Generates runtime code, which checks whether pointers and allocatable array references are defined and allocated. Should be used in conjunction with `-d{n}`. More...	OFF
`-CB` IA-32 compiler	Generates runtime code to check that array subscript and substring references are within declared bounds. Should be used in conjunction with `-d{n}`. More...	OFF
`-CS` IA-32 compiler	Generates runtime code that checks for consistent shape of intrinsic procedure. Should be used in conjunction with `-d{n}`. More...	OFF
`-CU` IA-32 compiler	Generates runtime code that causes a runtime error if variables are used without being initialized. Should be used in conjunction with `-d{n}`. More...	OFF
`-CV` IA-32 compiler	On entry to a subprogram, tests the correspondence between the actual arguments passed and the dummy arguments expected. Both calling and called code must be compiled with `-CV` for the checks to be effective. Should be used in conjunction with `-d{n}`. More...	OFF

`-cerrs[-]`	Enables/disables errors and warning messages to be printed in a terse format for diagnostic messages. More...	OFF
`-cm`	Suppresses all comment messages. More...	OFF
`-common_args`	Assumes by reference subprogram arguments may alias one another. More...	OFF
`-cpp{n}`	Same as `-fpp{n}`. More...	OFF
`-DD`	Compiles debugging statements indicated by the letter `D` in column 1 of the source code. More...	OFF
`-DX`	Compiles debugging statements indicated by the letters `X` in column 1 of the source code. More...	OFF
`-DY`	Compiles debugging statements indicated by the letters `Y` in column 1 of the source code. More...	OFF
`-d{n}` IA-32 compiler	Sets diagnostics level as follows: `-d0` - displays procname line `-d1` - displays local scalar variables `-d2` - local and common scalars `-d>2` - display first n elements of local and `COMMON` arrays, and all scalars. More...	`-d0`
`-D`*`name[`*`=text`*`]`*	Defines a macro name and associates it with the specified value. More...	OFF

`-dps, -nodps`	Enable (default) or disable DEC* parameter statement recognition. More...	`-dps`
`-dryrun`	Show driver tool commands but do not execute tools. More...	OFF
`-dynamic-linker(`*`file`*`)`	Specifies in *`file`* a dynamic linker of choice, rather than default.	OFF
`-e90`, `-e95`	Enable issuing of errors rather than warnings for features that are non-standard Fortran. More...	OFF
`-E`	Preprocesses the source files and writes the results to `_stdout`. If the file name ends with capital `F`, the option is treated as `-fpp1`. More...	OFF
`-EP`	Preprocesses the source files and writes the results to stdout omitting the `#line` directives. More...	OFF
`-extend_source`	Enables extended (132-character) source lines. Same as `-132`. More...	OFF
`-F`	Preprocesses the source files and writes the results to file. More...	OFF
`-falias`	Assumes aliasing in program. More...	ON
`-fno-alias`	Assumes no aliasing in program. More...	OFF
`-ffnalias`	Assumes aliasing within functions. More...	ON

`-fno-fnalias`	Assumes no aliasing within functions, but assumes aliasing across calls. More...	OFF
`-fcode_asm`	Produces assembly file listing with optional code byte annotations. More...	OFF
`-fsource_asm`	Produces assembly file listing with optional high-level source code annotations. More...	OFF
`-fverbose-asm`	Produces assembly file with compiler comments including compiler version and options. Enabled by default when producing an assembly file. More...	ON
`-fnoverbose-asm`	Produces assembly file without compiler comments. More...	OFF
`-fnsplit-` Itanium compiler	Disables function splitting, which is enabled by `-prof_use`. More...	OFF
`-FI`	Specifies that the source code is in fixed format. This is the default for source files with the file extensions `.for`, `.f`, or `.ftn`. More...	OFF
`-fp` IA-32 compiler	Disables the use of the `ebp` register in optimizations. Directs to use the `ebp`-based stack frame for all functions. More...	OFF

`-fpp{n}`	Enables the Fortran preprocessor (`fpp`) on all Fortran source files prior to compilation.
	n=0: disable CVF and #directives
	n=1: enable CVF conditional compilation and # directives;
		OFF

	when `fpp` runs, `-fpp1` is the default
	n=2: enable only # directives,
	n=3: enable only CVF conditional compilation directives.
	More...	

`-fp_port` IA-32 compiler	Rounds floating-point results at assignments and casts. Some speed impact. More...	OFF
`-FR`	Specifies that the source code is in Fortran free format. This is the default for source files with the `.f90` file extension. More...	OFF
`-ftz` Itanium compiler	Flushes denormal results to zero. More...	OFF
`-g`	Generates symbolic debugging information and line numbers in the object code for use by source-level debuggers. More...	OFF
`-help`	Prints help message. More...	OFF
`-i{2|4|8}`	Defines the default `KIND` for integer variables and constants to be 2, 4, and 8 bytes. More...	`-i4`
`-I`*`dir`*	Specifies an additional directory to search for include files whose names do not begin with a slash (/). More...	OFF
`-i_dynamic`	Sets dynamic linking of Intel-provided libraries as default. More...	OFF
`-implicitnone`	Sets `IMPLICIT NONE` as the default. Same as `-u`. More...	OFF

`-inline_debug_info`	Keep the source position of inlined code instead of assigning the call-site source position to inlined code. More...	OFF
`-ip`	Enables single-file interprocedural optimizations. More...	OFF
`-ip_no_inlining`	Disables full or partial inlining that would result from the `-ip` interprocedural optimizations. Requires `-ip` or `-ipo`. More...	ON
`-ip_no_pinlining` IA-32 compiler	Disables partial inlining. Requires `-ip` or `-ipo`. More...	OFF
`-IPF_fma[-]` Itanium® compiler	Enables/disables the contraction of floating-point multiply and add/ subtract operations into a single operation. More...	ON
`-IPF_fp_speculation`*mode* Itanium compiler	Sets the compiler to speculate on floating-point (fp) operations in one of the following modes: *`fast`*: speculate on fp operations; *`safe`*: speculate on fp operations only when it is safe; *`strict`*: enables the compiler's speculation on floating-point operations preserving floating-point status in all situations; same as `off` in the current version. *`off`*: disables the fp speculation. More...	`-IPF_fp_speculation` *`fast`*

`-IPF_flt_eval_method0` Itanium compiler	`-IPF_flt_eval_method0` directs the compiler to evaluate the expressions involving floating-point operands in the precision indicated by the program. More...	OFF
`-IPF_fltacc[-]` Itanium compiler	Enables/disables compiler optimizations that affect floating-point accuracy. More...	`-IPF_fltacc-`
`-ipo`	Enables interprocedural optimization across files. Compile all objects over entire program with multifile interprocedural optimizations. More...	OFF
`-ipo_c`	Optimizes across files and produces a multifile object file. This option performs optimizations as `-ipo`, but stops prior to the final link stage, leaving an optimized object file. More...	OFF
`-ipo_obj`	Forces the generation of real object files. Requires `-ipo`. More...	IA-32: OFF Itanium Compiler: ON
`-ipo_S`	Optimizes across files and produces a multifile assembly file. This option performs optimizations as `-ipo`, but stops prior to the final link stage, leaving an optimized assembly file. More...	OFF
`-ivdep_parallel` Itanium compiler	Indicates there is absolutely no loop-carried memory dependency in the loop where `IVDEP` directive is specified. More...	OFF
`-Kpic, -KPIC`	Generates position-independent code.	OFF

`-L`*`dir`*	Instructs linker to search *`dir`* for libraries. More...	OFF
`-l`*`name`*	Links with a library indicated in *`name`*. More...	OFF
`-list`	Prints a source listing to `stdout` (typically, your terminal screen) without contents of `include` files. More...	OFF
`-list -show`*`include`*	Prints a source listing to `stdout` with contents of `include` files expanded. More...	OFF
`-lowercase`	Sets the case of external linker symbols such as subroutine names to be lowercase characters. More...	ON
`-module[`*`path`*`]`, `-nomodule`	Specifies the directory where the module files (extension `.mod`) are placed. Omitting this option or specifying `-nomodule` results in placing the `.mod` files in the directory where the source files are being compiled. More...	`-nomodule`
`-mp`	Maintains declared floating point precision as well as conformance to the IEEE* 754 standards for floating-point arithmetic. Optimization is reduced accordingly. More...	OFF
`-mp1`	Restricts floating point precision to be closer to declared precision. Some speed impact, but less than `-mp`. More...	OFF

`-nbs`	Treats backslash (\) as a normal graphic character, not an escape character. More...	OFF
`-nobss_init`	Disables placement of zero-initialized variables in BSS (using DATA section) More...	OFF
`-nolib_inline`	Disables inline expansion of intrinsic functions. More...	ON
`-nologo`	Suppresses compiler version information. More...	ON
`-nus`	Disables appending an underscore to external subroutine names. More...	OFF
`-nus`*`file`*	Disables appending an underscore to subroutine names listed in *`file`*. More...	OFF
`-O, -O1, -O2` IA-32 compiler	Optimize for speed. Disable `-fp`. option. More...	OFF
`-O1` Itanium compiler	Optimizes to favor code size: turns off software pipelining to reduce code size. Enables the same optimizations as `-O` except for loop unrolling and software pipelining. More...	OFF
`-O2`	Optimizes for speed. Disables `-fp`. option. More...	ON
`-O0`	Disables optimizations. More...	OFF

`-O3`	Enables `-O2` option with more aggressive optimization, for example, loop transformation. Optimizes for maximum speed, but may not improve performance for some programs. More...	OFF
`-Ob{0|1|2}`	Controls the compiler's inline expansion. The amount of inline expansion performed varies as follows: `-Ob0`: disable inlining `-Ob1`: disables inlining unless `-ip` or `-Ob2` is specified. Enables inlining of functions. `-Ob2`: Enables inlining of any function. However, the compiler decides which functions are inlined. This option enables interprocedural optimizations and has the same effect as specifying the `-ip` option.	`-Ob1`
`-o`*`file`*	Indicates the executable file name in *`file`* ; for example, `-o`*`myfile`*. Combined with `-S`, indicates assembly listing file name. Combined with `-c`, indicates object file name. More...	OFF
`-onetrip`	Executes any DO loop at least once. (Identical to the `-1` option.) More...	OFF
`-openmp`	Enables the parallelizer to generate multithreaded code based on the OpenMP directives. This option implies that `-fpp` and `-auto` are ON. More...	OFF

`-openmp_report{0|1|2}`	Controls the OpenMP parallelizers diagnostic levels.	`-openmp_report1`
`-openmp_stubs`	Sets compilation of the OpenMP programs to be in sequential mode. The OpenMP directives are ignored and a stub OpenMP library is linked (sequentially).	OFF
`-opt_report` Itanium compiler	Generates optimizations report and directs to `stderr` unless `-opt_report_file` is specified. More...	OFF
`-opt_report_file` *`filename`* Itanium compiler	Specifies the *`filename`* to hold the optimizations report. More...	OFF
`-opt_report_level` {*`min`*|*`med`*|*`max`*} Itanium compiler	Specifies the detail level of the optimizations report. More...	`-opt_report_level`*`min`*
`-opt_report_phase`*`phase`* Itanium compiler	Specifies the optimization to generate the report for. Can be specified multiple times on the command line for multiple optimizations. More...	OFF
`-opt_report_help` Itanium compiler	Prints to the screen all available phases for `-opt_report_phase`. More...	OFF
`-opt_report_routine` *`routine_substring`* Itanium compiler	Generates reports from all routines with names containing the *`substring`* as part of their name. If not specified, reports from all routines are generated. More...	OFF
`-P`	Preprocesses the fpp files and writes the results to files named according to the compilers default file-naming conventions. More...	OFF

-pad, -nopad	Enables/disables changing variable and array memory layout. More...	-nopad
-pad_source	Enables the acknowledgment of blanks at the end of a line. More...	OFF
-parallel	Enables the auto-parallelizer to generate multithreaded code for loops that can be safely executed in parallel. More...	OFF
-par_threshold	Sets a threshold for the auto-parallelization of loops based on the probability of profitable execution of the loop in parallel, n=0 to 100. More...	n=75
-par_report{0|1|2|3}	Controls the auto-parallelizer's diagnostic levels. More...	-par_ report1
-pc32 -pc64 -pc80 IA-32 compiler	Enables floating-point significand precision control as follows: -pc32 to 24-bit significand -pc64 to 53-bit significand, and -pc80 to 64-bit significand More...	-pc64
-pg IA-32 compiler	Compile and link for function profiling with Linux gprof tool. More...	OFF
-posixlib	Enables linking to the POSIX* library (libPOSF90.a) in the compilation. More...	OFF
-prec_div IA-32 compiler	Disables floating point division-to-multiplication optimization resulting in more accurate division results. Slight speed impact. More...	OFF

`-prefetch[-]` IA-32 compiler	Enables or disables prefetch insertion (requires `-O3`). More...	ON
`-prof_dir`*`dir`*	Specifies the directory to hold profile information in the profiling output files, `*.dyn` and `*dpi`. More...	OFF
`-prof_gen`	Instruments the program for profiling: to get the execution count of each basic block. More...	OFF
`-prof_file`*`file`*	Specifies file name for profiling summary file. More...	OFF
`-prof_use`	Enables the use of profiling dynamic feedback information during optimization. More...	OFF
`-q`	Suppresses compiler output to standard error, `__stderr`. More...	OFF
`-Qdyncom"blk1,blk2,..."`	Enables dynamic allocation of given `COMMON` blocks at run time. More...	OFF
`-Qinstall`*`dir`*	Sets *`dir`* as a root directory for compiler installation. More...	OFF
`-Qlocation,`*`tool`*`,`*`path`*	Sets *`path`* as the location of the tool specified by *`tool`*. More...	OFF
`-Qloccom "blk1,blk2,..."`	Enables local allocation of given `COMMON` blocks at run time. More...	OFF
`-Qoption,`*`tool`*`,`*`opts`*	Passes the options, *`opts`*, to the tool specified by *`tool`*. More...	OFF

`-qp, -p`	Compile and link for function profiling with UNIX* `prof` tool. More...	OFF
`-r{8|16}`	Defines the `KIND` for real variables to be 8, or 16 bytes. By default, variables of type `REAL(4)` are used. `-r8`: change the size and precision of default `REAL` entities to `DOUBLE PRECISION`. Same as the `-autodouble`. `-r16`: change the size and precision of default `REAL` entities to `REAL (KIND=16)` More...	OFF
`-rcd` IA-32 compiler	Disables changing of rounding mode for floating-point-to-integer conversions. More...	OFF
`-S`	Produces an assembly output file. More...	OFF
`-safe_cray_ptr`	Specifies that Cray* pointers do not alias with other variables. More...	OFF
`-save`	Saves all variables (static allocation). Opposite of `-auto`. More...	OFF
`-scalar_rep[-]` IA-32 compiler	Enables or disables scalar replacement performed during loop transformations (requires `-O3`). More...	OFF
`-sox[-]` IA-32 compiler	Enables or disables (default) saving of compiler options and version in the executable. Itanium compiler: accepted for compatibility only. More...	OFF

`-shared`	Instructs the compiler to build a Dynamic Shared Object (DSO) instead of an executable. More...	OFF
`-static`	Sets static linking of the shared libraries (`.so`). More...	OFF
`-syntax`	Enables syntax check only. Same as `-y`. More...	OFF
`-Tf`*`file`*	Compiles *`file`* as a Fortran source. More...	OFF
`-tpp1` Itanium compiler	Targets optimization to the Intel® Itanium® processor for best performance. More...	OFF
`-tpp2` Itanium compiler	Targets optimization to the Intel® Itanium® 2 processor for best performance. Generated code is compatible with the Itanium processor. More...	ON
`-tpp{5|6|7}` IA-32 compiler	`-tpp5` optimizes for the Intel Pentium processor. `-tpp6` optimizes for the Intel Pentium Pro, Pentium II, and Pentium III processors. `-tpp7` optimizes for the Intel Pentium 4 and Xeon(TM) processor. More...	`-tpp7`
`-u`	Sets `IMPLICIT NONE` by default. Same as `-implicitnone`. More...	ON
`-U`*`name`*	Removes a defined macro specified by *`name`*; equivalent to an `#undef` preprocessing directive. More...	OFF

`-unroll[n]`	-Use n to set maximum number of times to unroll a loop. -Omit n to let the compiler decide whether to perform unrolling or not. -Use n = 0 to disable unroller. The Itanium compiler currently recognizes only n = 0; all other values are ignored. More...	ON
`-uppercase`	Sets the case of external linker symbols such as subroutine names to be uppercase characters. More...	OFF
`-us`	Appends (default) an underscore to external subroutine names. More...	ON
`-use_asm`	Produces objects through the assembler. More...	OFF
`-V`	Displays compiler version information. More...	OFF
`-v`	Shows driver tool commands and executes tools. More...	OFF
`-Vaxlib`	Enables linking to portability library (`libPEPCF90.a`) in the compilation. More...	OFF
`-vec _report{0|1|2|3|4|5}` IA-32 compiler	Controls amount of vectorizer diagnostic information as follows: n = 0: no information n = 1: indicate vectorized /non-vectorizerd loops n = 2: indicate vectorized /non-vectorized loops n = 3: indicate vectorized /non-vectorized loops and prohibit data dependence information n = 4: indicate non-vectorized loops	`-vec _report1`

	n = 5: indicate non-vectorized loops and the reason why they were not vectorized. More...	
-vms	Enables support for a certain set of extensions to Fortran that were introduced by Digital* VMS* and Compaq* Fortran compilers. More...	OFF
-w	Suppresses all warning messages. More...	OFF
-w90, -w95	Suppresses warning messages about Fortran features which are deprecated or obsoleted in Fortran 95. More...	OFF
-W{n}	Suppresses or displays all warning messages. n=0: suppresses all warnings n=1: displays all warnings (default). More...	-W1
-WB	On a bound check violation, issues a warning instead of an error. More...	OFF
-x{i|M|K|W} IA-32 compiler	Generates code that is optimized for a specific processor corresponding to one of codes: i, M, K, and W, but that will execute on any IA-32 processor. With this option, the resulting program may not run on processors older than the target specified. More...	OFF
-X	Removes standard directories from the include file search. More...	OFF
-y	Enables syntax check only. More...	OFF

`-zero`	Implicitly initializes to zero all data that is uninitialized. Used in conjunction with -save. More...	OFF
`-Zp{1|2|4|8|16}`	Specifies alignment constraint for structures on 1-, 2-, 4-, 8- or 16-byte boundary. More...	IA-32: `-Zp4` Itanium Compiler: `-Zp8`

Compiler Options by Functional Groups

Options entered on the command line change the compiler's default behavior, enable or disable compiler functionalities, and can improve the performance of your application. This section presents tables of compiler options grouped by Intel® Fortran Compiler functionality within these categories:

- Customizing Compilation Process Option Groups
- Language Conformance Option Groups
- Application Performance Optimizations

Key to the Tables

In each table:

- The functions are listed in alphabetical order
- The default status ON or default value is indicated; if not mentioned, the default is OFF
- The IA-32 or Itanium® architectures are indicated as follows:
  - not mentioned = used by both architectures
  - indicated in a row = used in the following rows exclusively by indicated architecture.

Each option group is described in detailed form in the sections of this documentation. Some options can be viewed as belonging to more than one group; for example, option `-c` that tells compiler to stop at creating an object file, can be viewed as monitoring either compilation or linking. In such cases, the options are mentioned in more than one group.

Customizing Compilation Process Options

Alternate Tools and Locations

Option	Description	Default
`-Qlocation,`*`tool`*`,`*`path`*	Enables you to specify a *`path`* as the location of the specified *`tool`* (such as the assembler, linker, preprocessor, and compiler). See Specifying Alternate Tools and Locations.	OFF
`-Qoption,`*`tool`*`,`*`opts`*	Passes the options specified by *`opts`* to a *`tool`*, where *`opts`* is a comma-separated list of options. See Passing Options to Other Tools.	OFF

Preprocessing

See the Preprocessing section for more information.

Option	Description	Default
`-cpp{n}`	Same as `-fpp{n}`.	OFF
`-D`*`name`* *`[`*`=text`*`]`	Defines the macro name and associates it with the specified value. The default (`-D`*`name`*) defines a macro with value =1.	OFF
`-E`	Directs the preprocessor to expand your source file and write the result to standard output.	OFF
`-EP`	Same as `-E` but does not include #line directives in the output.	OFF
`-F`	Preprocesses to an indicated file. Directs the preprocessor to expand your source module and store the result in a file in the current directory.	OFF

`-fpp{n}`	Uses the `fpp` preprocessor on Fortran source files.
	n=0: disable CVF and `#directives`
	n=1: enable CVF conditional compilation and # directives; when `fpp` runs, `-fpp1` is the default
	n=2: enable only `#directives`,
	n=3: enable only CVF conditional compilation directives.
		OFF
		
		(`-fpp1` when `fpp` runs)

`-I`*`dir`*	Adds directory *`dir`* to the include and module file search path.	OFF
`-P`	Directs the preprocessor to expand your source file and store the result in a file in the current directory.	OFF

`-U`*`name`*	Eliminates any definition *`name`* currently in effect.	OFF
`-X`	Removes standard directories from the include file search path.	OFF

Compiling

See detailed Compiling section.

Option	Description	Default
`-0f_check` IA-32 only	Avoid incorrect decoding of some `0f` instructions; enable the patch for the Pentium® `0f` erratum.	OFF
`-align`	Analyzes and reorders memory layout for variables and arrays.	`-align`
`-noalign`	Disables `-align`.	OFF
`-c`	Compile to object only (`.o`), do not link.	OFF
`-dynamic-linker`*`file`*	Specifies in *`file`* a dynamic linker of choice, rather than default.	OFF
`-falias`	Assumes aliasing in program.	ON
`-fno-alias`	Assumes no aliasing in program..	OFF
`-ffnalias`	Assumes aliasing within functions.	ON
`-fno-fnalias`	Assumes no aliasing within functions, but assumes aliasing across calls.	OFF
`-fp` IA-32 only	Disables using `ebp` as general purpose register in optimizations. Directs to use the `ebp`-based stack frame for all functions.	OFF
`-ftz` Itanium®-based systems	Flushes denormal results (floating-point values smaller than smallest normalized floating-point number) to zero. Use this option when the denormal values are not critical to application behavior.	OFF
`-I`*`dir`*	Adds directory *`dir`* to the include and module file search path.	OFF
`-Kpic, -KPIC`	Generate position-independent code.	OFF
`-module[path]`, `-nomodule`	Specifies the directory where the module files (extension `.mod`) are placed. Omitting this option or specifying `-nomodule` results in placing the `.mod` files in the directory where the source files are being compiled.	`-nomodule`
`-nobss_init`	Disable placement of zero-initialized variables in BSS (using Data).	OFF

`-p, -qp`	Compile and link for function profiling with UNIX* prof tool.	OFF
`-pg` IA-32 only	Compile and link for function profiling with Linux* gprof tool.	OFF
`-Qinstall,`*`dir`*	Sets root directory of compiler installation, indicated in *`dir`* to contain all compiler install files and subdirectories.	OFF
`-S`	Produce assembly file named `file.asm` with optional code or source annotations. Do not link.	OFF
`-sox[-]` IA-32 only	Enable (default) or disable saving of compiler options and version in the executable.	OFF
`-Tf`*`file`*	Compile *`file`* as Fortran source.	OFF
`-use_asm`	Produces objects through the assembler.	OFF
`-Zp{n}`	Specifies alignment constraint for structures on n-byte boundary (n = 1, 2, 4, 8, 16). The `-Zp16` option enables you to align Fortran structures such as common blocks.	IA-32: `-Zp4` Itanium® Compiler: `-Zp8`

Linking

See detailed Linking section.

Option	Description	Default
`-Bdynamic`	Used with `-l`*`name`* (see below), enables dynamic linking of libraries at run time. Compared to static linking, results in smaller executables.	OFF
`-Bstatic`	Enables linking a user's library statically.	
`-c`	Compile to object only (`.o`), do not link.	OFF
`-C90`	Link with alternate I/O library for mixed output with the C language.	OFF
`-dynamic-linker`*`file`*	Specifies in *`file`* a dynamic linker of choice, rather than default.	OFF
`-i_dynamic`	Enables to link Intel-provided libraries dynamically.	OFF
`-L`*`dir`*	Instructs linker to search *`dir`* for libraries.	OFF

`-l`*`name`*	Link with a library indicated in name.	OFF
`-p, -qp`	Compile and link for function profiling with UNIX prof tool.	OFF
`-pg` IA-32 only	Compile and link for function profiling with Linux gprof tool.	OFF
`-posixlib`	Enables linking with POSIX* library.	OFF
`-shared`	Instructs the compiler to build a Dynamic Shared Object (DSO) instead of an executable.	OFF
`-static`	Enables static linking of libraries.	OFF
`-Vaxlib`	Enable linking with portability library.	OFF

Compilation Output

See the Specifying Compilation Output section for more information.

Option	Description	Default
`-c`	Compile to object only (`.o`), do not link.	OFF
`-fcode-asm`	Produces assembly file with optional code byte information.	OFF
`-fsource-asm`	Produces assembly file with optional high-level source code information.	OFF
`-fverbose-asm`	Produces assembly file with compiler comments including compiler version and options used. Enabled by default when producing an assembly file.	OFF
`-fnoverbose-asm`	Produces assembly file without compiler comments.	OFF
`-list`	Prints a source listing to `stdout`.	OFF
`-list -show`*`include`*	Prints a source listing to `stdout` with contents of `include` files expanded.	OFF
`-o`*`file`*	Produces the executable file name specified in *`file`*; for example, `-o`*`myfile`*. Combined with `-S`, indicates assembly listing file name. Combined with `-c`, indicates object file name.	OFF
`-S`	Produce assembly file named `file.asm` with optional code or source annotations. Do not link.	OFF

Debugging

See the Debugging section for more information.

Option	Description	Default
`-DD`	Compiles debug statements indicated by a `D` or a `d` in column 1; if this option is not set these lines are treated as comments	OFF
`-DX`	Compiles debug statements indicated by a `X` (not an `x`) in column 1; if this option is not set these lines are treated as comments.	OFF
`-DY`	Compiles debug statements indicated by a `Y` (not a `y`) in column 1; if this option is not set these lines are treated as comments.	OFF
`-inline_debug_info`	Keeps the source position of inline code instead of assigning the call-site source position to inlined code.	OFF
`-g`	Produces symbolic debug information in the object file.	OFF
`-y, -syntax`	Both perform syntax check only.	OFF

Libraries

See detailed section on Libraries.

Option	Description	Default
`-C90`	Link with alternate I/O library for mixed output with the C language.	OFF
`-i_dynamic`	Enables to link Intel-provided libraries dynamically.	OFF
`-L`*`dir`*	Instructs linker to search *`dir`* for libraries.	OFF
`-l`*`name`*	Links with the library indicated in *`name`*.	OFF
`-posixlib`	Link with POSIX* library.	OFF
`-shared`	Instructs the compiler to build a Dynamic Shared Object (DSO) instead of an executable.	OFF
`-static`	Enables to link shared libraries (`.so`) statically.	OFF
`-Vaxlib`	Link with portability library.	OFF

Diagnostics and Messages

See Diagnostics and Messages section for more information.

Runtime Diagnostics (IA-32 Compiler only)

Option	Description	Default
`-C`	Equivalent to: (`-CA`, `-CB`, `-CS`, `-CU`, `-CV`) extensive runtime diagnostics options.	OFF
`-CA`	Use in conjunction with `-d{n}`. Checks for `nil` pointers/allocatable array references at runtime.	OFF
`-CB`	Use in conjunction with `-d{n}`. Generates runtime code to check that array subscript and substring references are within declared bounds.	OFF
`-CS`	Use in conjunction with `-d{n}`. Generates runtime code that checks for consistent shape of intrinsic procedure.	OFF
`-CU`	Use in conjunction with `-d{n}`. Generates runtime code that causes a runtime error if variables are used without being initialized.	OFF
`-CV`	Use in conjunction with `-d{n}`. On entry to a subprogram, tests the correspondence between the actual arguments passed and the dummy arguments expected. Both calling and called code must be compiled with `-CV` for the checks to be effective.	OFF
`-d{n}`	Set the level of diagnostic messages, `n=0`, `1`, `2`, `>2`	`-d0`

Compiler Information Messages

Option	Description	Default
`-nologo`	Disables the display of the compiler version (or sign-on) message: compiler ID, version, copyright years.	OFF
`-help`	You can print a list and brief description of the most useful compiler driver options by specifying the `-help` option on the command line.	OFF
`-V`	Displays compiler version information.	OFF
`-v`	Shows driver tool commands and executes tools.	OFF
`-dryrun`	Shows driver tool commands, but does not execute tools.	OFF

Comment and Warning Messages

Option	Description	Default
`-cm`	Suppresses all comment messages.	OFF
`-cerrs[-]`	Enables/disables (default) a terse format for diagnostic messages, for example: `"file", line no : error message`	`-cerrs`
`-w`	Suppresses all warning messages.	OFF
`-w90`, `-w95`	Suppresses warning messages about Fortran features which are deprecated or obsoleted in Fortran 95.	OFF
`-W{n}`	Suppresses or displays all warning messages generated by preprocessing and compilation. n=0: suppresses all warnings n=1: displays all warnings (default).	`-W1`
`-WB`	On a bound check violation, issues a warning instead of an error (accommodates old FORTRAN code, in which array bounds of dummy arguments were frequently declared as 1.)	OFF

Error Messages

Option	Description	Default
`-e90`, `-e95`	Enable issuing of errors rather than warnings for features that are non-standard Fortran.	OFF
`-q`	Suppresses compiler output to standard error, `_stderr`. When `-q` is specified with `-bd`, then only fatal error messages are output to `_stderr`.	OFF

Language Conformance Options

Data Type

See more details in Setting Data Types and Sizes.

Option	Description	Default
`-autodouble`	Sets the default size of real numbers to 8 bytes; same as `-r8`.	OFF
`-i{2|4|8}`	Specifies that all quantities of `integer` type and unspecified `kind` occupy two bytes. All quantities of `logical` type and unspecified `kind` will also occupy two bytes. All logical constants and all small integer constants occupy two bytes. `-i4`: All `integer` and `logical` types of unspecified `kind` will occupy four bytes. `-i8`: All `integer` and `logical` types of unspecified `kind` will occupy eight bytes.	`-i4`
`-r{4|8|16}`	Defines the `KIND` for real variables in 4 (default), 8, and 16 bytes. `-r8`: change the size and precision of default `REAL` entities to `DOUBLE PRECISION`. Same as the `-autodouble`. `-r16`: change the size and precision of default `REAL` entities to `REAL (KIND=16)`.	`-r4`

Source Program

See more details in Source Program Features.

Option	Description	Default
`-1`	Same as `-onetrip`.	OFF
`-132`	Enables fixed form source lines to contain up to 132 characters.	OFF
`-ansi_alias[-]`	Enables (default) or disables assumption of the program's ANSI conformance. Provides cross-platform compatibility .	`-ansi_alias`
`-dps`, `-nodps`	Enables (default) or disables DEC* parameter statement recognition.	`-dps`
`-extend_source`	Enables extended (132-character) source lines. Same as `-132`.	OFF
`-FI`	Specifies that all the source code is in fixed format; this is the default except for files ending with the suffix `.f`, `.ftn`, `.for`.	OFF

`-FR`	Specifies that all the source code is in Fortran free format; this is the default for files ending with the suffix `.f90`.	OFF
`-lowercase`	Controls the case of routine names and external linker symbols to all lowercase characters.	`ON`
`-nbs`	Treats backslash (\) as a normal graphic character, not an escape character. This may be necessary when transferring programs from non-UNIX* environments, for example from VAX* VMS*. For the effects of the escape character, see the Escape Characters.	OFF
`-nus[`*`file`*`]`	Do not append an underscore to subroutine names listed in *`file`*. Useful when linking with C routines.	OFF
`-onetrip`	Compiles `DO` loops at least once if reached (by default, Fortran 95 `DO` loops are not performed at all if the upper limit is smaller than the lower limit). Same as `-1`.	OFF
`-pad_source`	Enforces the acknowledgment of blanks at the end of a line.	OFF
`-uppercase`	Maps routine names to all uppercase characters. **Note** Do not use this option in combination with `-Vaxlib` or `-posixlib`.	OFF
`-vms`	Enables support for extensions to Fortran that were introduced by Digital* VMS Fortran compilers. The extensions are as follows: • The compiler enables shortened, apostrophe-separated syntax for parameters in I-O statements. • The compiler assumes that the value specified for `RECL` in an `OPEN` statement is	OFF

	given in words rather than bytes. This option also implies `-dps` (on by default).	

Arguments and Variables

See more details in Setting Arguments and Variables.

Option	Description	Default
`-align`	Analyze and reorder memory layout for variables and arrays.	`-align`
`-noalign`	Disables `-align`.	OFF
`-auto`	Makes all local variables `AUTOMATIC`. Causes all variables to be allocated on the stack, rather than in local static storage.	OFF
`-auto_scalar`	Causes scalar variables of rank 0, except for variables of the `COMPLEX` or `CHARACTER` types, to be allocated on the stack, rather than in local static storage. Enables the compiler to make better choices concerning variables that should be kept in registers during program execution. On by default.	ON
`-common_args`	Assumes "by reference" subprogram arguments may have aliases of one another.	OFF
`-implicitnone`	Enables the default `IMPLICIT NONE`.	OFF
`-safe_cray_ptr`	Specifies that Cray pointers do not alias with other variables.	OFF
`-save`	Forces the allocation of all variables in static storage. If a routine is invoked more than once, this option forces the local variables to retain their values from the first invocation terminated. Opposite of `-auto`.	OFF
`-u`	Enables the default `IMPLICIT NONE`. Same as `-implicitnone.`	OFF
`-zero`	Initializes all data to zero. It is most commonly used in conjunction with `-save`.	OFF

Common Blocks

See Allocating Common Blocks for more information.

Option	Description	Default
`-Qdyncom"blk1, blk2, ..."`	Dynamically allocates `COMMON` blocks at run time.	OFF
`-Qloccom"blk1, blk2, ..."`	Enables local allocation of given `COMMON` blocks at run time.	OFF

Application Performance Optimizations Options

Setting Optimization Level

See the Optimization Levels section for more information.

Option	Description	Default
`-O1`	IA-32 compiler: Optimizes for speed. Disables `-fp` option. Itanium® compiler: Turns off software pipelining to reduce code size. Optimizes to favor code size. Enables the same optimizations as `-O2` except for loop unrolling. Generally, `-O2` is recommended over `-O1`.	OFF
`-O, -O2`	Optimizes for speed. Disables `-fp`. option.	`-O2`
`-O3`	Enables `-O2` option with more aggressive optimization and sets high-level optimizations, including loop transformation, OpenMP, and prefetching. High-level optimizations use the properties of source code constructs such as loops and arrays in applications written in high-level programming languages. Optimizes for maximum speed, but may not improve performance for some programs.	OFF
`-O0`	Disables optimizations `-O1`, `-O2` and `-O3`. Enables option `-fp`.	OFF

Floating-point Arithmetic Precision

See Floating-point Arithmetic Optimizations for more information.

Option	Description	Default
`-fp_port` IA-32 only	Rounds floating-point results at assignments and casts. Some speed impact.	OFF
`-IFP_fma[-]` Itanium®-based systems	Enables/disables the contraction of floating-point multiply and add/subtract operations into a single operation.	`-IFP_fma`
`-IPF_fp_speculation`*`mode`* Itanium-based systems	Sets the compiler to speculate on fp operations in one of the following modes: *`fast`*: speculate on fp operations; *`safe`*: speculate on fp operations only when it is safe; *`strict`*: enables the compiler's speculation on floating-point operations preserving floating-point status in all situations; same as `off` in the current version. *`off`*: disables fp speculation.	`-IPF_fpc64_speculation`*`fast`*
`-IPF_flt_eval_method0` Itanium-based systems	`-IPF_flt_eval_method0` directs the compiler to evaluate the expressions involving floating-point operands in the precision indicated by the program. (`-IPF_flt_eval_method2` is not supported in the current version.)	OFF
`-IFP_fltacc[-]` Itanium-based systems	Enables/disables the compiler to apply optimizations that affect floating-point accuracy.	`-IFP_fltacc-`
`-mp`	Maintains declared precision and ensures that floating-point arithmetic conforms more closely to the ANSI and IEEE* 754 standards. See details in the Maintaining and Restricting FP Arithmetic Precision.	OFF

`-mp1`	Restricts floating-point precision to be closer to declared precision. Some speed impact, but less than `-mp`. See details in the Maintaining and Restricting FP Arithmetic Precision.	OFF
`-pc{32|64|80}` IA-32 only	Enables floating-point significand precision control as follows: `-pc32` to 24-bit significand `-pc64` to 53-bit significand (Default) `-pc80` to 64-bit significand	`-pc64`
`-prec_div` IA-32 only	Disables floating point division-to-multiplication optimization resulting in more accurate division results. Slight speed impact.	OFF
`-rcd` IA-32 only	Disables changing of rounding mode for floating-point-to-integer conversions.	OFF

Processor Dispatch Support

See Processor Dispatch Extensions Support for more information.

Option	Description	Default
`-tpp1` Itanium®-based systems	Targets optimization to the Intel® Itanium® processor for best performance.	OFF
`-tpp2` Itanium-based systems	Targets optimization to the Intel® Itanium® 2 processor for best performance. Generated code is compatible with the Itanium processor.	`-tpp2`
`-tpp5` IA-32 only	Optimizes for the Intel Pentium® processor. Enables best performance for Pentium® processor	OFF
`-tpp6` IA-32 only	Optimizes for the Intel Pentium Pro, Pentium II, and Pentium III processors. Enables best performance for the above processors.	OFF
`-tpp7` IA-32 only	Optimizes for the Pentium 4 and Xeon(TM) processors. Requires the RedHat version 7.1 and support of Streaming SIMD Extensions 2. Enables best performance for Pentium 4 processor	`-tpp7`

`-ax{i|M|K|W}` IA-32 only	Generates, in a single binary, code specialized to the extensions specified by the codes: `i` Pentium Pro, Pentium II processors `M` Pentium with MMX(TM) technology processor `K` Pentium III processor (Streaming SIMD Extensions) `W` Pentium 4 and Xeon processors In addition, `-ax` generates IA-32 generic code. The generic code is usually slower.	OFF
`-x{i|M|K|W}` IA-32 only	Generate specialized code to run exclusively on the processors supporting the extensions indicated by the codes: `i` Pentium Pro, Pentium II processors `M` Pentium with MMX technology processor `K` Pentium III processor `W` Pentium 4 and Xeon processors	OFF

Interprocedural Optimizations

See Interprocedural Optimizations (IPO) section for more information.

Option	Description	Default
`-ip`	Enables single-file interprocedural optimizations. Enhances inline function expansion.	OFF
`-ip_no_inlining`	Disables full or partial inlining that would result from the `-ip` interprocedural optimizations. Requires `-ip` or `-ipo`.	OFF
`-ip_no_pinlining` IA-32 only	Disables partial inlining. Requires `-ip` or `-ipo`.	OFF
`-ipo`	Enables interprocedural optimization across files. Compile all objects over entire program with multifile interprocedural optimizations. Enhances multifile optimization; multifile inline function expansion, interprocedural constant and function characteristics propagation, monitoring module-level static variables; dead code elimination.	OFF
`-ipo_c`	Optimizes across files and produces a multifile object file. This option performs the same optimizations as `-ipo`, but stops prior to the final link stage, leaving an optimized object file.	OFF

`-ipo_obj`	Forces the generation of real object files. Requires `-ipo`.	OFF
`-ipo_S`	Optimizes across files and produces a multifile assembly file. This option performs the same optimizations as `-ipo`, but stops prior to the final link stage, leaving an optimized assembly file.	OFF
`-inline_debug_info`	Preserve the source position of inlined code instead of assigning the call-site source position to inlined code.	OFF
`-Ob{0|1|2}`	Controls the compiler's inline expansion. The amount of inline expansion performed varies as follows: `-Ob0`: disable inlining `-Ob1`: disables inlining unless `-ip` or `-Ob2` is specified. Enables inlining of functions. `-Ob2`: Enables inlining of any function. However, the compiler decides which functions are inlined. This option enables interprocedural optimizations and has the same effect as specifying the `-ip` option.	`-Ob1`
`-nolib_inline`	Disables inline expansion of intrinsic functions.	OFF

Profile-guided Optimizations

See detailed Profile-guided Optimizations section.

Option	Description	Default
`-fnsplit[-]` Itanium® compiler	Disables function splitting, which is enabled by `-prof_use`.	OFF
`-prof_dir`*`dir`*	Specifies the directory to hold profile information in the profiling output files, `*.dyn` and `*.dpi`.	OFF
`-prof_file`*`file`*	Specifies file name for profiling summary file.	OFF
`-prof_gen`	Instruments the program for profiling: to get the execution count of each basic block.	OFF

`-prof_use`	Enables the use of profiling dynamic feedback information during optimization. Profiles the most frequently executed areas and increases effectiveness of IPO.	OFF

High-level Language Optimizations

See detailed High-level Language Optimizations (HLO) section.

Option	Description	Default
`-ivdep_parallel` Itanium® compiler	Indicates there is absolutely no loop-carried memory dependency in the loop where `IVDEP` directive is specified.	OFF
`-prefetch[-]` IA-32 only	Enables or disables prefetch insertion (requires `-O3`). Reduces the wait time; optimum use is determined empirically.	`-prefetch`
`-scalar_rep[-]` IA-32 only	Enables (default) or disables scalar replacement performed during loop transformations (requires `-O3`). Eliminates all loads and stores of that variable Increases register pressure.	`-scalar_rep`
`-unroll[n]`	n: set maximum number of times to unroll a loop n omitted: compiler decides whether to perform unrolling or not. n = 0: disables unroller. Eliminates some code; hides latencies; can increase code size. For Itanium®-based applications, `-unroll[o]` is used only for compatibility.	`-unroll`

Parallelization

See detailed Parallelization section.

Option	Description	Default
`-openmp`	Enables the parallelizer to generate multi-threaded code based on the OpenMP* directives. Enables parallel execution on both uni- and multiprocessor systems. Requires `-fpp`.	OFF

`-openmp_report{0|1|2}`	Controls the OpenMP parallelizer's diagnostic levels: 0 - no information 1 - loops, regions, and sections parallelized (default) 2 - same as 1 plus master construct, single construct, etc.	`-openmp_report1`
`-openmp_stubs`	Enables to compile OpenMP programs in sequential mode. The OpenMP directives are ignored and a stub OpenMP library is linked (sequentially).	OFF
`-parallel`	Enables the auto-parallelizer to generate multithreaded code for loops that can be safely executed in parallel.	OFF
`-par_report{0|1|2|3}`	Controls the auto-parallelizer's diagnostic levels: 0 - no information 1 - successfully auto-parallelized loops 2 - successfully and unsuccessfully auto-parallelized loops 3 - same as 2 plus additional information about any proven or assumed dependences inhibiting auto-parallelization.	`-par_report1`
`-par_threshold{n}`	Sets a threshold for the auto-parallelization of loops based on the probability of profitable execution of the loop in parallel, n=0 to 100.	n=75

Vectorization (IA-32 only)

See detailed Vectorization section.

Option	Description	Default
`-ax{i|M|K|W}` IA-32 only	Generates, on a single binary, code specialized to the extensions specified by the codes: `i` Pentium Pro, Pentium II processors `M` Pentium with MMX technology processor `K` Pentium III processor `W` Pentium 4 and Xeon(TM) processors In addition, `-ax` generates IA-32 generic code. The generic code is usually slower.	OFF

	Note: `-axi` is not a vectorizer option.	
`-x{i|M|K|W}` IA-32 only	Generate specialized code to run exclusively on the processors supporting the extensions indicated by the codes: `i` Pentium Pro, Pentium II processors `M` Pentium with MMX technology processor `K` Pentium III processor `W` Pentium 4 and Xeon processors **Note:** `-xi` is not a vectorizer option.	OFF
`-vec_report{0|1|2|3|4|5}` IA-32 only	Controls the diagnostic messages from the vectorizer as follows: n = 0: no information n = 1: indicates vectorized /non-vectorizerd loops n = 2: indicates vectorized /non-vectorized loops n = 3: indicates vectorized /non-vectorized loops and prohibit data dependence information n = 4: indicates non-vectorized loops n = 5: indicates non-vectorized loops and the reason why they were not vectorized	`-vec_report1`

Optimization Reports (Itanium® Compiler)

See detailed Optimizer Report Generation.

These options are implemented with Itanium®-based systems only.

Option	Description	Default
`-opt_report`	Generates optimizations report and directs to `stderr` unless `-opt_report_file` is specified.	OFF
`-opt_report_file`*`filename`*	Specifies the *`filename`* to hold the optimizations report.	OFF
`-opt_report_level` *`min|med|max`*`}`	Specifies the detail level of the optimizations report.	`-opt_report_level`*`min`*
`-opt_report_phase`*`phase`*	Specifies the optimization to generate the report for. Can be specified multiple times on the command line for multiple optimizations.	OFF

`-opt_report_help`	Prints to the screen all available phases for `-opt_report_phase`.	OFF
`-opt_report_routine` *`routine_substring`*	Generates reports from all routines with names containing the *substring* as part of their name. If not specified, reports from all routines are generated.	OFF

Windows* to Linux* Options Cross-reference

This section provides cross-reference table of the Intel® Fortran Compiler options used on the Windows* and Linux* operating systems. The options described can be used for compilations targeted to either IA-32 or Itanium®-based applications or both. See Conventions Used in the Options Quick Guide Tables.

- Options specific to IA-32 architecture
- Options specific to the Itanium® architecture

All other options are available for both IA-32 and Itanium architectures.

Note
The table is based on the alphabetical order of compiler options for Linux.

Note
The value in the Default column is used for both Windows and Linux operating systems unless indicated otherwise.

Windows Option	Linux Option	Description	Default
`/QI0f[-]` IA-32 only	`-0f_check` IA-32 only	Enables a software patch for Pentium® processor `0f` erratum.	OFF
`/1`	`-1`	Executes any `DO` loop at least once.	OFF
`/4L {72|80|132}`	`-72, -80, -132`	Specifies 72, 80 or 132 column lines for fixed form source only. The compiler might issue a warning for non-numeric text beyond 72 for the `-72` option.	`/4L72` `-72`
`/align`	`-align`	Analyzes and reorders memory layout for variables and arrays.	ON

/align-	-noalign	Disables .-align	OFF
/Qansi_alias[-]	-ansi_alias [-]	Enables (default) or disables assumption of the programs ANSI conformance.	ON
/Qauto	-auto	Makes all local variables AUTOMATIC.	OFF
/Qautodouble	-autodouble	Sets the default size of real numbers to 8 bytes; same as -r8.	OFF
/Qauto_scalar	-auto_scalar	Makes scalar local variables AUTOMATIC.	ON
/Qax{i|M|K|W} IA-32 only	-ax{i|M|K|W} IA-32 only	Generates code that is optimized for a specific processor, but that will execute on any IA-32 processor. Compiler generates multiple versions of some routines, and chooses the best version for the host processor at runtime. supporting the extensions indicated by processor-specific codes i (Pentium® Pro), M (Pentium with MMX(TM) technology), K (Pentium III), and W (Pentium 4 and Xeon(TM)).	OFF
None	-Bdynamic	Used with -l*name* (see in this table), enables dynamic linking of libraries at run time. Compared to static linking, results in smaller executables.	OFF
None	-Bstatic	Enables linking a user's library statically.	OFF
/c	-c	Stops the compilation process after an object file (.o) has been generated.	OFF

/C IA-32 only	-C IA-32 only	Enable extensive runtime error checking. Equivalent to: -CA, -CB, -CS, -CU, or -CV runtime diagnostics options.	OFF
/CA IA-32 only	-CA IA-32 only	Generates code check at runtime to ensure that referenced pointers and allocatable arrays are not nil. Should be used in conjunction with -d{n}.	OFF
/CB IA-32 only	-CB IA-32 only	Generates code to check that array subscript and substring references are within declared bounds. Should be used in conjunction with -d{n}.	OFF
/CS IA-32 only	-CS IA-32 only	Generates code to check the shapes of array arguments to intrinsic procedures. Should be used in conjunction with -d{n}.	OFF
/CU IA-32 only	-CU IA-32 only	Generates code that causes a runtime error if variables are used without being initialized. Should be used in conjunction with -d{n}.	OFF
/CV IA-32 only	-CV IA-32 only	On entry to a subprogram, tests the correspondence between the actual arguments passed and the dummy arguments expected. Both calling and called code must be compiled with -CV for the checks to be effective. Should be used in conjunction with -d{n}.	OFF
/C90	-C90	Links with an alternative I/O library (libCEPCF90.a) that supports mixed input and output with C on the standard streams.	OFF

/cerrs[-]	-cerrs[-]	Enables/disables errors and warning messages to be printed in a terse format.	Windows ON Linux: OFF
/cm	-cm	Suppresses all comment messages.	OFF
/Qcommon_args	-common_args	Assumes by reference subprogram arguments may have aliases of one another.	OFF
/Qcpp[n]	-cpp[n]	Same as -fpp.	OFF
/Qd_lines	-DD	Compiles debugging statements indicated by the letter D in column 1 of the source code.	OFF
/Qdx_lines	-DX	Compiles debugging statements indicated by the letters X in column 1 of the source code.	OFF
/Qdy_lines	-DY	Compiles debugging statements indicated by the letters Y in column 1 of the source code.	OFF
/d{n} IA-32 only	-d{n} IIA-32 only	Sets diagnostics level as follows: -d0 - displays procedure name and line -d1 - displays local scalar variables -d2 - local and common scalars -d>2 - display first n elements of local and COMMON arrays, and all scalars.	-d0
/D*name [={#|text}]*	-D*name [={#|text}]*	Defines a macro name and associates it with the specified value.	OFF
/Qdps[-]	-dps, -nodps	Enable (default) or disable DEC* parameter statement recognition.	Windows ON Linux: -dps
None	-dryrun	Show driver tool commands but do not execute tools.	OFF
None	-dynamic-linker (*file*)	Specifies in *file* a dynamic linker of choice, rather than default.	OFF

/E	-E	Preprocesses the source files and writes the results to _stdout. If the file name ends with capital F, the option is treated as fpp.	OFF
/4{Y|N}s	-e90, -e95	Enables/disables issuing of errors rather than warnings for features that are non-standard Fortran.	OFF
/EP	-EP	Preprocesses the source files and writes the results to stdout omitting the #line directives.	OFF
/Qextend_source	-extend_source	Enables extended (132-character) source lines. Same as -132.	OFF
/P	-F	Preprocesses the source files and writes the results to file.	OFF
/Oa-	-falias	Assumes aliasing in program.	ON
/Oa	-fno-alias	Assumes no aliasing in program.	OFF
/Ow-	-ffnalias	Assumes aliasing within functions.	ON
/Ow	-fno-fnalias	Assumes no aliasing within functions, but assumes aliasing across calls.	OFF
/FAc	-fcode-asm	Produces assembly file with optional code byte annotations.	OFF
/FAs	-fsource-asm	Produces assembly file with optional high-level source code annotations.	OFF
None	-fverbose-asm	Produces assembly file with compiler comments including compiler version and options. Enabled by default when producing an assembly file.	ON
None	-fnoverbose-asm	Produces assembly file without compiler comments.	OFF

/FI	-FI	Specifies that the source code is in fixed format. This is the default for source files with the file extensions .for, .f, or .ftn.	OFF
/Qfnsplit-	-fnsplit- Itanium-based systems	Disables function splitting, which is enabled by -prof_use.	OFF
/Oy- IA-32 only	-fp IA-32 only	Disables the use of the ebp register in optimizations. Directs to use the ebp-based stack frame for all functions.	OFF
/Qfp_port	-fp_port IA-32 only	Rounds floating-point results at assignments and casts. Some speed impact.	OFF

/Qfpp{n}	-fpp{n}	Enables the Fortran preprocessor (fpp) on all Fortran source files prior to compilation.
		n=0 disable CVF and # directives, equivalent to no fpp.
		n=1 enable CVF conditional compilation and # directives; when fpp runs, -fpp1 is the default
		n=2 enable only # directives
		n=3 enable only CVF conditional directives
			OFF

/FR	-FR	Specifies that the source code is in Fortran 95 free format. This is the default for source files with the .f90 file extensions.	OFF
-Qftz Itanium-based systems	-ftz Itanium-based systems	Flushes denormal results to zero.	OFF
/ZI, /Z7	-g	Generates symbolic debugging information and line numbers in the object code for use by source-level debuggers.	OFF

/help	-help	Prints help message.	OFF
/4I{2|4|8}	-i{2|4|8}	Defines the default KIND for integer variables and constants in 2, 4, and 8 bytes.	/4I4 -i4
None	-i_dynamic	Enables to link Intel-provided libraries dynamically.	OFF
/I*dir*	-I*dir*	Specifies an additional directory to search for include and module files whose names do not begin with a slash (/).	OFF
/4{Y|N}d	-implicitnone	Enables/disables the IMPLICIT NONE.	OFF
/Qinline_debug_info	-inline_debug_info	Keep the source position of inline code instead of assigning the call-site source position to inlined code.	OFF
/Qip	-ip	Enables single-file interprocedural optimizations within a file.	OFF
/Qip_no_inlining	-ip_no_inlining	Disables full or partial inlining that would result from the -ip interprocedural optimizations. Requires -ip or -ipo.	ON
/Qip_no_pinlining IA-32 only	-ip_no_pinlining IA-32 only	Disables partial inlining. Requires -ip or -ipo.	OFF
/QIPF_fma[-] Itanium-based systems	-IPF_fma[-] Itanium-based systems	Enables/disables the contraction of floating-point multiply and add/subtract operations into a single operation.	ON
/QIPF_fp_speculation*mode* Itanium-based systems	-IPF_fp_speculation*mode* Itanium-based systems	Sets the compiler to speculate on fp operations in one of the following modes: *fast*: speculate on fp operations; *safe*: speculate on fp operations only when it is safe; *strict*: enables the	*mode*=*fast*

		compiler's speculation on floating-point operations preserving floating-point status in all situations; *off*: disables the fp speculation.	
/QIPF_flt_eval_method0 Itanium-based systems	-IPF_flt_eval_method0 Itanium-based systems	-IPF_flt_eval_method0 directs the compiler to evaluate the expressions involving floating-point operands in the precision indicated by the program.	OFF
/QIPF_fltacc[-] Itanium-based systems	-IPF_fltacc[-] Itanium-based systems	Enables/disables the compiler to apply optimizations that affect floating-point accuracy.	OFF
/Qipo	-ipo	Enables interprocedural optimization across files. Compile all objects over entire program with multifile interprocedural optimizations.	OFF
/Qipo_c	-ipo_c	Optimizes across files and produces a multifile object file. This option performs optimizations as -ipo, but stops prior to the final link stage, leaving an optimized object file.	OFF
/Qipo_obj	-ipo_obj	Forces the generation of real object files. Requires -ipo.	IA-32: OFF Itanium Compiler ON
/Qipo_S	-ipo_S	Optimizes across files and produces a multifile assembly file. This option performs optimizations as -ipo, but stops prior to the final link stage, leaving an optimized assembly file.	OFF
/Qivdep_parallel Itanium-based systems	-ivdep_parallel Itanium-based systems	Indicates there is absolutely no loop-carried memory dependency in the loop where IVDEP directive is specified.	OFF

None	`-Kpic, -KPIC`	Generates position-independent code.	OFF
None	`-L`*`dir`*	Instructs linker to search dir for libraries.	OFF
None	`-l`*`name`*	Links with the library indicated in *`name`*.	
`/list`	`-list`	Prints a source listing to `stdout` (typically, your terminal screen) without contents of `INCLUDE` files.	OFF
`/list /show:`*`include`*	`-list -show`*`include`*	Prints a source listing to `stdout` with contents of `include` files expanded.	OFF
`/Qlowercase`	`-lowercase`	Changes routine names to lowercase characters which are uppercase by default. (Linux: also controls the external symbol names in lowercase.)	Windows OFF Linux: O
`/Fm`*`filename`*	`None`	Instructs the linker to produce a map file.	OFF
`/module[`*`path`*`], /nomodule`	`-module [`*`path`*`], -nomodule`	Specifies the directory where the module files (extension `.mod`) are placed. Omitting this option or specifying `-nomodule` results in placing the `.mod` files in the directory where the source files are being compiled.	`-nomodu`
`/Op[-]`	`-mp`	Maintains declared floating-point precision as well as conformance to the IEEE 754 standards for floating-point arithmetic. Optimization is reduced accordingly.	OFF
`/Qprec`	`-mp1`	Restricts floating floating-point precision to be closer to declared precision. Some speed impact, but less than `-mp`.	OFF

/nbs	-nbs	Treats backslash (\) as a normal graphic character, not an escape character.	OFF
/Qnobss_init	-nobss_init	Disables placement of zero-initialized variables in BSS (using DATA section)	OFF
/Oi-	-nolib_inline	Disables inline expansion of intrinsic functions.	ON
/nologo	-nologo	Suppresses compiler version information.	OFF
None	-nus	Disables appending an underscore to external subroutine names.	OFF
/us	None	Append an underscore to external subroutine names	OFF
/Od	-O0	Disables optimizations.	OFF
/O2	-O, -O1, -O2	Optimize for speed., but disable some optimizations that increase code size for a small speed benefit. For Itanium compiler, -O1 turns off software pipelining to reduce code size.	ON
/O3	-O3	Enables -O2 option with more aggressive optimization, for example, loop transformation. Optimizes for maximum speed, but may not improve performance for some programs.	OFF
/Ob{0|1|2}	-Ob{0|1|2}	Controls the compiler's inline expansion. The amount of inline expansion performed varies as follows: -Ob0: disable inlining -Ob1: disables inlining unless -ip or -Ob2 is specified. Enables inlining of functions. -Ob2: Enables inlining of	-Ob1

		any function. However, the compiler decides which functions are inlined. This option enables interprocedural optimizations and has the same effect as specifying the `-ip` option.	
`/Fo`*`filename`*	`-o`*`file`*	Name the object file or directory for multiple files.	OFF
`/Fa`*`filename`*	None	Name assembly file or directory for multiple files.	
`/Fe`*`filename`*	None	Name executable file or directory.	
`/Qonetrip`	`-onetrip`	Executes any `DO` loop at least once. (Identical to the `-1` option.).	OFF
`/Qopenmp`	`-openmp`	Enables the parallelizer to generate multithreaded code based on the OpenMP* directives. This option implies that `-fpp` is ON.	OFF
`/Qopenmp_report{0|1|2}`	`-openmp_report {0|1|2}`	Controls the OpenMP parallelizers diagnostic levels.	`-openmp_report`
`/Qopenmp_stubs`	`-openmp_stubs`	Enables to compile OpenMP programs in sequential mode. The OpenMP directives are ignored and a stub OpenMP library is linked (sequentially).	OFF
`/Qopt_report` Itanium-based systems	`-opt_report` Itanium-based systems	Generates optimizations report and directs to `stderr` unless `-opt_report_file` is specified.	OFF
`/Qopt_report_file`*`filename`* Itanium-based systems	`-opt_report_file`*`filename`* Itanium-based systems	Specifies the *`filename`* to hold the optimizations report.	OFF
`/Qopt_report_help` Itanium-based systems	`-opt_report_help` Itanium-based systems	Prints to the screen all available phases for `-opt_report_phase`.	OFF

/Qopt_report_level {*min*|*med*|*max*} Itanium-based systems	-opt_report_level {*min*|*med*|*max*} Itanium-based systems	Specifies the detail level of the optimizations report.	-opt_report-level *min*
/Qopt_report_phase*phase* Itanium-based systems	-opt_report_phase*phase* Itanium-based systems	Specifies the optimization to generate the report for. Can be specified multiple times on the command line for multiple optimizations.	OFF
/Qopt_report_routine*routine*_*substring* Itanium-based systems	-opt_report_routine*routine*_*substring* Itanium-based systems	Generates reports from all routines with names containing the *substring* as part of their name. If not specified, reports from all routines are generated.	OFF
/P	-P	Preprocesses the fpp files and writes the results to files named according to the compilers default file-naming conventions.	OFF
/Qpad[-]	-pad	Enables/disables changing variable and array memory layout.	OFF
/Qpad_source	-pad_source	Enforces the acknowledgment of blanks at the end of a line.	OFF
/Qparallel	-parallel	Enables the auto-parallelizer to generate multi-threaded code for loops that can be safely executed in parallel.	OFF
/Qpar_report{0|1|2|3}	-par_report {0|1|2|3}	Controls the auto-parallelizer's diagnostic levels.	-par_report
/Qpar_threshold{n}	-par_threshold {n}	Sets a threshold for the auto-parallelization of loops based on the probability of profitable execution of the loop in parallel, n=0 to 100. This option is used for loops whose computation work volume cannot be determined at compile-time.	n=75

/Qpc{32|64|80} IA-32 only	-pc32 -pc64 -pc80 IA-32 only	Enables floating-point significand precision control as follows: -pc32 to 24-bit significand -pc64 to 53-bit significand -pc80 to 64-bit significand	/Qpc64 -pc64
None	-pg IA-32 only	Compile and link for function profiling with Linux gprof tool.	OFF
/4{Y|N}posixlib	-posixlib	Enables/disables (Windows) linking to the POSIX* library (libPOSF90.a) in the compilation.	OFF
/Qprec_div IA-32 only	-prec_div IA-32 only	Disables floating point division-to-multiplication optimization resulting in more accurate division results. Slight speed impact.	OFF
/Qprefetch[-] IA-32 only	-prefetch[-] IA-32 only	Enables or disables prefetch insertion (requires -O3).	OFF
/Qprof_dir*dir*	-prof_dir*dir*	Specifies the directory to hold profile information in the profiling output files, *.dyn and *dpi.	OFF
/Qprof_gen	-prof_gen	Instruments the program for profiling: to get the execution count of each basic block.	OFF
/Qprof_file*file*	-prof_file*file*	Specifies file name for profiling summary file.	OFF
/Qprof_use	-prof_use	Enables the use of profiling dynamic feedback information during optimization.	OFF
/q	-q	Suppresses compiler output to standard error, __stderr.	OFF
/Qdyncom*com1*[,*com2*]	-Qdyncom *com1*[,*com2*]	Enables dynamic allocation of given COMMON blocks at run time.	OFF
None	-Qinstall,*dir*	Sets dir as a root directory for compiler installation.	OFF

/Qlocation,*tool*,*path*	-Qlocation,*tool*,*path*	Specifies an alternate version of a tool located at path.	OFF
/Qloccom,*com1*[,*com2*,...*comn*]	-Qloccom,*com1*[,*com2*,...*comn*]	Enables local allocation of given COMMON blocks at run time.	OFF
/Qoption,*tool*,*opts*	-Qoption,*tool*,opts	Passes the options, opts, to the tool specified by tool.	OFF
None.	-qp, -p	Compile and link for function profiling with UNIX* prof tool.	OFF
/4R{4|8|16}	-r{4|8|16}	Defines the KIND for real variables in 4 (default), 8, and 16 bytes. -r8: change the size and precision of default REAL entities to DOUBLE PRECISION. Same as the -autodouble. -r16: change the size and precision of default REAL entities to REAL (KIND=16)	-r8
/Qrcd IA-32 only	-rcd IA-32 only	Disables changing of rounding mode for floating-point-to-integer conversions.	OFF
/S	-S	Produces an assembly output file with optional code.	OFF
/Qsafe_cray_ptr	-safe_cray_ptr	Specifies that Cray* pointers do not alias with other variables.	OFF
/Qsave	-save	Saves all variables (static allocation). Opposite of -auto.	OFF
/Qscalar_rep[-] IA-32 only	-scalar_rep[-] IA-32 only	Enables or disables scalar replacement performed during loop transformations (requires -O3).	OFF

/Qsox[-]	-sox[-] IA-32 only	Enables or disables (default) saving of compiler options and version in the executable. Itanium compiler: accepted for compatibility only.	OFF
None	-shared	Instructs the compiler to build a Dynamic Shared Object (DSO) instead of an executable.	OFF
None	-static	Enables to link shared libraries (.so) statically.	OFF
None	-syntax	Enables syntax check only. Same as -y.	OFF
/Tf*file*	-Tf*file*	Compile file as Fortran source.	OFF
/G1 Itanium-based systems	-tpp1 Itanium-based systems	Targets optimization to the Intel® Itanium® processor for best performance.	OFF
/G2 Itanium-based systems	-tpp2 Itanium-based systems	Targets optimization to the Intel® Itanium® 2 processor for best performance. Generated code is compatible with the Itanium processor.	/G2 -tpp2
/G{5|6|7} IA-32 only	-tpp{5|6|7} IA-32 only	-tpp5 optimizes for the Intel Pentium processor. -tpp6 optimizes for the Intel Pentium Pro, Pentium II, and Pentium III processors. -tpp7 optimizes for the Intel Pentium 4 and Xeon processors; requires the support of Streaming SIMD Extensions 2.	/G7 -tpp7
/4{Y|N}d	-u	Sets IMPLICIT NONE by default.	ON
/U*name*	-U*name*	Removes a defined macro; equivalent to an #undef preprocessing directive.	OFF

/Qunroll[n]	-unroll[n]	- Use n to set maximum number of times to unroll a loop. - Omit n to let the compiler decide whether to perform unrolling or not. - Use n = 0 to disable unroller. The Itanium compiler currently uses only n = 0; all other values are NOPs.	ON
/Quppercase	-uppercase	Changes routine names to all uppercase characters.	Windows ON Linux: OFF
None	-use_asm	Generates an assembly file and tells the assembler to generate the object file.	OFF
/V*string*	-V	Displays compiler version information.	OFF
None	-v	Shows driver tool commands and executes tools.	OFF
/4{Y|N}portlib	-Vaxlib	Enables/disables linking to portlib library (libPEPCF90.a) in the compilation.	OFF
/Qvec_report{n} IA-32 only	-vec_report{n} IA-32 only	Controls amount of vectorizer diagnostic information as follows: n = 0: no information n = 1: indicate vectorizer loops n = 2: same as n = 1 plus non-vectorizer loops n = 3: same as n = 1 plus dependence information. n = 4: indicate non-vectorized loops n = 5: indicate non-vectorized loops and and the reason why they were not vectorized.	n = 1

/Qvms	-vms	Enables support for I/O and DEC extensions to Fortran that were introduced by Digital* VMS and Compaq* Fortran compilers.	OFF
/w	-w	Suppresses all warning messages.	OFF
/W0	-W0	Disables display of warnings.	OFF
/W1	-W1	Displays warnings.	ON
/w90, /w95	-w90, -w95	Suppresses warning messages about Fortran features which are deprecated or obsoleted in Fortran 95.	OFF
/WB	-WB	Issues a warning about compile time bound check violation.	OFF
/Qx{i|M|K|W} IA-32 only	-x{i|M|K|W} IA-32 only	Generates processor-specific code corresponding to one of codes: i, M, K, and W while also generating generic IA-32 code. This differs from -ax{n} in that this targets a specific processor. With this option, the resulting program may not run on processors older than the target specified. i = Pentium Pro & Pentium II processor information M = MMX(TM) instructions K = streaming SIMD extensions W = Pentium® 4 and Xeon new instructions.	OFF
/X	-X	Removes standard directories from the include file search.	OFF
None	-y	Enables syntax check only.	OFF
/Qzero	-zero	Implicitly initializes to zero all data that is uninitialized otherwise. Used in conjunction with -save.	OFF

/Zp{1|2|4|8|16}	-Zp {1|2|4|8|16}	Specifies alignment constraint for structures on 1-, 2-, 4-, 8- or 16-byte boundary.	Windows OFF Linux: IA-32: -Zp4 Itanium Compiler -Zp8

Getting Started with the Intel® Fortran Compiler

Invoking Intel® Fortran Compiler

The Intel® Fortran Compiler has the following variations:

- Intel® Fortran Compiler for 32-bit Applications is designed for IA-32 systems, and its command is `ifc`. The IA-32 compilations run on any IA-32 Intel processor and produce applications that run on IA-32 systems. This compiler can be optimized specifically for one or more Intel® IA-32 processors, from Intel® Pentium® to Pentium 4 to Celeron(TM) and Xeon(TM) processors.

- Intel® Fortran Itanium® Compiler for Itanium®-based Applications, or native compiler, is designed for Itanium architecture systems, and its command is `efc`. This compiler runs on Itanium-based systems and produces Itanium-based applications. Itanium-based compilations can only operate on Itanium-based systems.

You can invoke compiler from:

- compiler command line
- makefile command line

Invoking from the Compiler Command Line

To invoke the Intel® Fortran Compiler from the command line requires these steps :

1. Set the environment variables
2. Issue the compiler command, `ifc` or `efc`

Setting the Environment Variables

Set the environment variables to specify locations for the various components. The Intel Fortran Compiler installation includes shell scripts that you can use to set environment variables. From the command line, execute the shell script that corresponds to your installation. With the default compiler installation, these scripts are located at:

IA-32 systems:
`/opt/intel/compiler60/ia32/bin/ifcvars.sh`

Itanium®-based systems:
`/opt/intel/compiler60/ia64/bin/efcvars.sh`

Running the Shell Scripts

To run the `ifcvars.sh` script on IA-32, enter the following on the command line:

prompt`>. /opt/intel/compiler70/ia32/bin/ifcvars.sh`

If you want the `ifcvars.sh` to run automatically when you start Linux*, edit your `.bash_profile` file and add the following line to the end of your file:

```
# set up environment for Intel compiler ifc
. /opt/intel/compiler70/ia32/bin/ifcvars.sh
```

The procedure is similar for running the `efcvars.sh` shell script on Itanium®-based systems.

Command Line Syntax

The command for invoking the compiler depends on what processor architecture you are targeting the compiled file to run on, IA-32 or Itanium®-based applications. The following describes how to invoke the compiler from the command line for each targeted architecture.

- **Targeted for IA-32 architecture:**
  prompt`>ifc [options] file1.f [file2.f . . .]`

- **Targeted for Itanium® architecture:**
  prompt`>efc [options] file1.f [file2.f . . . .]`

Note
Throughout this manual, where applicable, command line syntax is given for both IA-32- and Itanium-based compilations as seen above.

`options`	Indicates one or more command-line options. The compiler recognizes one or more letters preceded by a hyphen (-) as an option. Some options take arguments in the form of filenames, strings, letters, or numbers. Except where otherwise noted, you can enter a space between the option and its argument (s) or you can combine them.
`file1, file2 . . .`	Indicates one or more files to be processed by the compilation system. You can specify more than one ***`file`***. Use a space as a delimiter for multiple files. See Compiler Input Files.

Note
Specified options on the command line apply to all files. For example, in the following command line, the `-c` and `-w` options apply to both files `x.f` and `y.f`:

```
prompt>ifc -c x.f -w y.f

prompt>efc -c x.f -w y.f
```

Command Line with make

To specify a number of files with various paths and to save this information for multiple compilations, you can use makefiles. To use a makefile to compile your input files using the Intel® Fortran Compiler, make sure that `/usr/bin` and `/usr/local/bin` are on your path.

If you use the C shell, you can edit your `.cshrc` file and add

```
setenv PATH  /usr/bin:/usr/local/bin:<your path>
```

Then you can compile as

```
make -f  <Your makefile>
```

where `-f` is the `make` command option to specify a particular makefile.

For some versions of make, a default Fortran compiler macro `F77` is available. If you want to use it, you should provide the following settings in the startup file for your command shell:

- **On an IA-32 system:** `F77 ifc`
- **On an Itanium®-based system:** `F77 efc`

Input Files

The Intel® Fortran Compiler interprets the type of each input file by the filename extension; for example, `.a`, `.f`, `.for`, .o, and so on.

Filename	Interpretation	Action
filename.a	object library	Passed to `ld`.
filename.f	Fortran source	Compiled by Intel® Fortran Compiler, assumes fixed-form source.
filename.ftn	Fortran source	Compiled by Intel Fortran Compiler; assumes fixed form source.
filename.for	Fortran source	Compiled by Intel Fortran Compiler; assumes fixed form source.
filename.fpp	Fortran fixed-form source	Preprocessed by the Intel Fortran preprocessor `fpp`; then compiled by the Intel Fortran Compiler.

filename.f90	Fortran 90/95 source	Compiled by Intel Fortran Compiler; free-form source.
filename.F	Fortran fixed-form source	Passed to preprocessor (`fpp`) and then compiled by the Intel Fortran compiler
filename.s	**IA-32** assembly file	Passed to the assembler.
filename.s	**Itanium®** assembly file	Passed to the Intel Itanium assembler.
filename.o	Compiled object file	Passed to `ld(1)`.

You can use the compiler configuration file `ifc.cfg` for IA-32 or `efc.cfg` for Itanium-based applications to specify default directories for input libraries and for work files. To specify additional directories for input files, temporary files, libraries, and for the assembler and the linker, use compiler options that specify output file and directory names.

Default Behavior of the Compiler

By default, the compiler generates executable file(s) of the input file(s) and performs the following actions:

- Searches for all files, including library files, in the current directory
- Passes options designated for linking as well as user-defined libraries to the linker
- Displays error and warning messages
- Supports the extended ANSI standard for the Fortran language.
- Performs default settings and optimizations using options summarized in the Default Behavior of the Compiler Options section.
- For IA-32 applications, the compiler uses `-tpp7` option to optimize the code for the Pentium® 4 and Xeon(TM) processor; for Itanium®-based applications, the compiler uses `-tpp2` option to optimize the code for the Itanium® 2 processor.

For unspecified options, the compiler uses default settings or takes no action. If the compiler cannot process a command-line option, that option is passed to the linker.

Default Behavior of the Compiler Options

If you invoke the Intel® Fortran Compiler without specifying any compiler options, the default state of each option takes effect. The following tables summarize the options whose default status is ON as they are required for Intel Fortran Compiler default operation. The tables group the options by their functionality.

Per your application requirement, you can disable one or more options.

For the default states and values of all options, see the Compiler Options Quick Reference Alphabetical table. The table provides links to the sections describing the functionality of the options. If an option has a default value, such value is indicated. If an option includes an optional minus [-], this option is ON by default.

The following tables list all options that compiler uses for its default execution.

Data Setting and Language Conformance

Default Option	Description
-72	-72,-80,-132 specifies the column length for fixed form source only. The compiler might issue a warning for non-numeric text beyond 72 for the -72 option.
-align	Analyzes and reorders memory layout for variables and arrays.
-ansi_alias[-]	Enables assumption of the program's ANSI conformance.
-r4	Specifies the size of the real numbers to four bytes. -r{8|16} works the same as -align only with specific settings: specifies the size of real numbers to 8 (IA-32 systems, same as -autodouble) or 16 bytes for Itanium® compiler.
-auto_scalar	Makes scalar local variables AUTOMATIC.
-dps	Enables DEC* parameter statement recognition.
-i4	-i{2|4|8} defines the default KIND for integer variables and constants in 2, 4, and 8 bytes.
-lowercase	Controls the case of routine names and external linker symbols to all lowercase characters.
-pad	Enables changing variable and array memory layout.
-pc64 IA-32 only	-pc{32|64|80} enables floating-point significand precision control as follows: -pc32 to 24-bit significand, -pc64 to 53-bit significand, and -pc80 to 64-bit significand.
-save	Saves all variables in static allocation. Disables -auto, that is, disables setting all variables AUTOMATIC.
-u	Sets IMPLICIT NONE.
-us	Appends an underscore to external subroutine names.

IA-32: `-Zp4` Itanium compiler: `-Zp8`	`-Zp{n}` specifies alignment constraint for structures on 1-, 2-, 4-, 8-, or 16-byte boundary. To disable, use `-align-`.

Optimizations

Default Option	Description
`-fp` IA-32 only	Disables the use of the `ebp` register in optimizations. Directs to use the `ebp`-based stack frame for all functions.
`-ip_no_inlining`	Disables full or partial inlining that would result from the `-ip` interprocedural optimizations. Requires `-ip` or `-ipo`.
`-IPF_fma` Itanium® compiler	Enables the contraction of floating-point multiply and add/subtract operations into a single operation.
`-IPF_fp_speculation`*`fast`* Itanium compiler	Sets the compiler to speculate on floating-point operations. `-IPF_fp_speculation`*`off`* disables this optimization.
`-ipo_obj` Itanium compiler	Forces the generation of real object files. Requires `-ipo`. IA-32 systems: OFF
`-O, -O1, -O2`	Optimize for maximum speed.
`-Ob1`	Disables inlining unless `-ip` or `-Ob2` is specified.
`-openmp_report1`	Indicates loops, regions, and sections parallelized.
`-opt_report_level`*`min`*	Specifies the minimal level of the optimizations report.
`-par_report1`	Indicates loops successfully auto-parallelized.
`-tpp2` Itanium compiler	Optimizes code for the Intel® Itanium® 2 processor for Itanium-based applications. Generated code is compatible with the Itanium processor.
`-tpp7` IA-32 only	Optimizes code for the Intel® Pentium® 4 and Xeon(TM) processor for IA-32 applications.

`-unroll`	`-unroll[`*n*`]`: omit *n* to let the compiler decide whether to perform unrolling or not (default). Specify *n* to set maximum number of times to unroll a loop. The Itanium compiler currently uses only *n* = 0, `-unroll0` (disabled option) for compatibility.
`-vec_report1`	Indicates loops successfully vectorized.

Compilation

Default Option	Description
`-falias`	Assumes aliasing in program.
`-ffnalias`	Assumes aliasing within functions.
`-fverbose-asm`	Produces assembly file with compiler comments including compiler version and options used.
`-fpp1` (for preprocessor only)	When preprocessor runs, enables CVF conditional and # directives.
`-sox-`	Disables saving of compiler options and version in the executable. For Itanium-based systems, accepted for compatibility only.

Messages and Diagnostics

Default Option	Description
`-cerrs`	Enables errors and warning messages to be printed in a terse format. To disable, use `-cerrs-`.
`-d0`	Displays only the procedure name and the number of the line at which the failure occurred.
`-W1`	Displays warnings.

Disabling Default Options

To disable an option, use one of the following as applies:

- Generally, to disable one or a group of optimization options, use `-O0 option`. For example:

  IA-32 applications:

  prompt>**ifc -O2 -O0 *input_file(s)***

Itanium-based applications:

```
prompt>efc -O2 -O0 input_file(s)
```

Note
The `-O0` option is part of a mutually-exclusive group of options that includes `-O0`, `-O`, `-O1`, `-O2`, and `-O3`. The last of any of these options specified on the command line will override the previous options from this group.

- To disable options that include optional "-" shown as `[-]`, use that version of the option in the command line, for example: `-align-`.
- To disable options that have `{n}` parameter, use n=0 version, for example: `-unroll0`.

Note
If there are enabling and disabling versions of switches on the line, the last one takes precedence.

Resetting Default Data Types

To reset data type default options, you need to indicate a new option which overrides the default setting. For example:

IA-32 applications:

```
prompt>ifc -i2 input_file(s)
```

Itanium-based applications:

```
prompt>efc -i2 input_file(s)
```

Option `-i2` overrides default option `-i4`.

Default Libraries and Tools

For the libraries provided with Intel® Fortran Compiler, see IA-32 compiler libraries list and Itanium® compiler libraries list.

The default tools are summarized in the table below.

Tool	Default	Provided with Intel Fortran Compiler
IA-32 Assembler	Linux* Assembler, `as`	No
Itanium® Assembler	Intel® Itanium® Assembler	Yes
Linker		No

You can specify alternate to default tools and locations for preprocessing, compilation, assembly, and linking.

Assembler

By default, the compiler generates an object file directly without calling the assembler. However, if you need to use specific assembly input files and then link them with the rest of your project, you can use an assembler for these files.

IA-32 Applications

For 32-bit applications, Linux supplies its own assembler, `as`. For Itanium-based applications, to compile to assembly files and then use an assembler to produce executables, use the Itanium assembler, `ias`.

Itanium®-based Applications

If you need to assemble specific input files and link them to the rest of your project object files, produce object files using Intel® Itanium® assembler with `ias` command. For example, if you want to link some specific input file to the Fortran project object file, do the following:

1. Issue command using `-S` option to generate an assembly code file, `file.s`.

```
prompt>efc -S -c file.f
```

2. To assemble the `file.s` file, call Itanium® assembler with this command:

```
prompt>ias -Nso -p32 -o file.o file.s
```

where the following assembler options are used:

`-Nso` suppresses sign-on message

`-p32` enables defining 32-bit elements as relocatable data elements. Kept for backward compatibility

`-o`*`file`* indicates the output object file name

The above command generates an object file, `file.o`, which you can link with the object file of the whole project.

Linker

The compiler calls the system linker, `ld(1)`, to produce an executable file from object files. The linker searches the environment variable `LD_LIBRARY_PATH` to find available libraries.

Compilation Phases

To produce the executable file `filename`, the compiler performs by default the compile and link phases. When invoked, the compiler driver determines which compilation phases to perform based on the extension of the source filename and on the compilation options specified in the command line.

The table that follows lists the compilation phases and the software that controls each phase.

Phases	Software	IA-32 or Itanium® Architecture
Preprocess (Optional)	`fpp`	Both
Compile	`f90com`	Both
Assemble	`ias`	Itanium architecture
Link	`ld`	Both

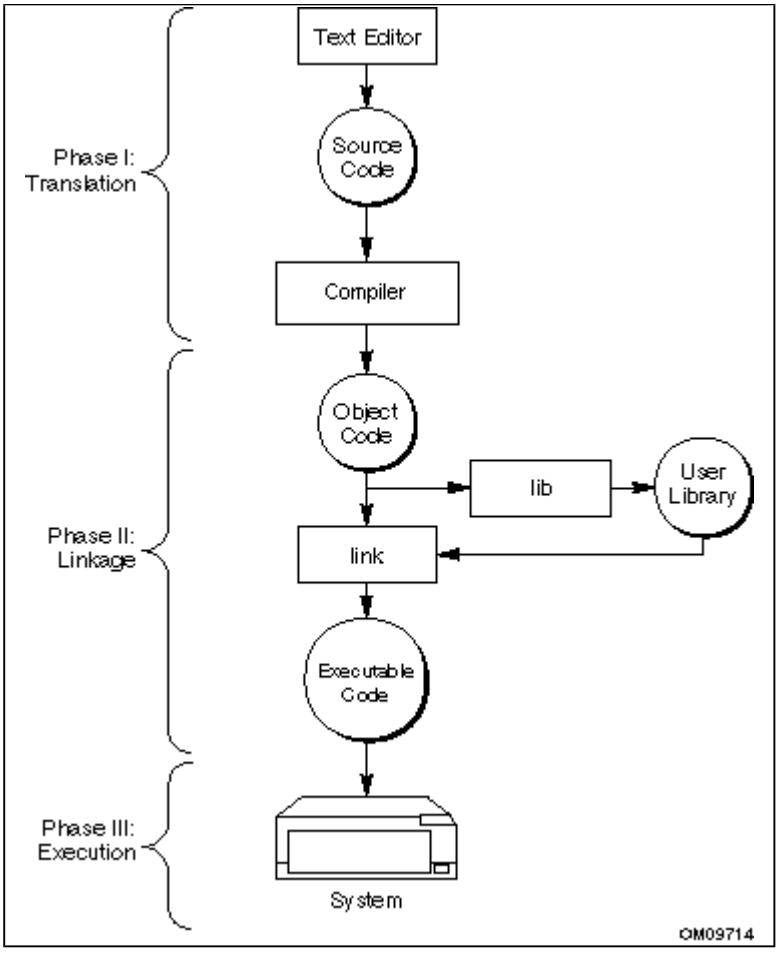
The compiler passes object files and any unrecognized filename to the linker. The linker then determines whether the file is an object file (`.o`) or a library (`.a`). The compiler driver handles all types of input files correctly, thus it can be used to invoke any phase of compilation.

Application Development Cycle

The relationship of the compiler to system-specific programming support tools is presented in the Application Development Cycle diagram.

The compiler processes Fortran language source and generates object files. You decide the input and output by setting options when you run the compiler. The figure shows how the compiler fits into application development environment.

Application Development Cycle


Text Editor
Source Code
Compiler
Object Code
lib
User Library
link
Executable Code
System
Phase I: Translation
Phase II: Linkage
Phase III: Execution
OM09714

Customizing Compilation Environment

You can customize the compilation process of your Fortran programs with the Fortran Compilation Environment (FCE) included with the Intel® Fortran Compiler. FCE provides a methodology of handling compilation according to the size and structure of your program. In addition, the FCE provides a methodology for code reusability and other automated features. The modular approach also facilitates several levels of use, from short programs to complex and large-scale projects.

To customize the environment used during compilation, you can specify the variables, options, and files as follows:

- Environment variables to specify paths where the compiler searches for special files such as libraries and "include" files
- Configuration files to use the options with each compilation
- Response files to use the options and files for individual projects
- Include Files to use for your application

Environment Variables

There are a number of environment variables that control the compiler's behavior. These environment variables can be set in the startup file for your command shell, or your `.login` file. Alternatively, you can invoke the setting variables script before running the compiler.

You can also set the `PATH` and `LD_LIBRARY_PATH` in your `.login` file only, there will no longer be any need to execute the setting variables script before running the compiler.

The following variables are relevant to your compilation environment.

`EFCCFG`	Specifies the configuration file that the compiler should use instead of the default configuration file for the Itanium® compiler.
`IFCCFG`	Specifies the configuration file that the compiler should use instead of the default configuration file for the IA-32 compiler.
`F_UFMTENDIAN`	Specifies the numbers of the units to be used for little-endian-to-big-endian conversion purposes.
`LD_LIBRARY_PATH`	Specifies the directory path for the libraries loaded at run-time.

PATH	Specifies the directory path for the compiler executable files. Enables the compiler to search for libraries or include files. You can establish these variables in the startup file for your command shell. You can use the `env` command to determine what environment variables you already have set.
TMP	Specifies the directory in which to store temporary files. If the directory specified by `TMP` does not exist, the compiler places the temporary files in the current directory.

Configuration File Environment Variables

`IFCCFG` and `EFCCFG` environment variables specify the configuration file that the compiler should use instead of the default configuration file. The default configuration files are `ifc.cfg` for the 32-bit Intel Fortran compiler and `efc.cfg` for the Itanium compiler in the `/bin` directory, and by default, the compiler always picks up the `.cfg` file from the same directory where the compiler executable resides. However, if the user needs to use a configuration file in a different location, they can use the `IFCCFG` or `EFCCFG` environment variable and assign the directory and filename of the `.cfg` file that needs to be picked up by the compiler.

Configuration Files

To decrease the time when entering command line options and ensure consistency of often-used command-line entries, use the configuration files. You can insert any valid command-line options into the configuration file. The compiler processes options in the configuration file in the order they appear followed by the command-line options that you specify when you invoke the compiler.

> **Note**
>
> Be aware that options placed in the configuration file will be included each time you run the compiler. If you have varying option requirements for different projects, see Response Files.

These files can be added to the directory where Intel® Fortran Compiler is installed.

Examples that follow illustrate sample `.cfg` files. The pound (#) character indicates that the rest of the line is a comment.

IA-32 applications: `ifc.cfg`

You can put any valid command-line option into this file.

```
## Sample ifc.cfg file for IA-32
applications
##
## Define preprocessor macro MY_PROJECT.
-Dmy_project
##
## Set extended-length source lines.
-132
##
## Set maximum floating-point significand
precision.
-pc80
##
## Link with alternate I/O library for
mixed output with the
## C language.
-C90
```

Itanium®-based applications: `efc.cfg`

```
## Sample efc.cfg file for Itanium®-based
applications
##
## Define preprocessor macro MY_PROJECT.
-Dmy_project
##
## Enable extended-length source lines.
-132
##
## Link with alternate I/O library for
mixed output with the
## C language.
-C90
```

Response Files

Use response files to specify options used during particular compilations for particular projects, and to save this information in individual files. Response files are invoked as an option on the command line. Options specified in a response file are inserted in the command line at the point where the response file is invoked.

Response files are used to decrease the time spent entering command-line options, and to ensure consistency by automating command-line entries. Use individual response files to maintain options for specific projects; in this way you avoid editing the configuration file when changing projects.

You can place any number of options or filenames on a line in the response file. Several response files can be referenced in the same command line.

The syntax for using response files is as follows :

IA-32 applications:

```
prompt>ifc @response_filename
```

```
prompt>ifc @response_filename1 @response_filename2
```

Itanium®-based applications:

```
prompt>efc @response_filename
```

```
prompt>efc @response_filename1 @response_filename2
```

 **Note**

An "at" sign (@) must precede the name of the response file on the command line.

Include Files

Include files are brought into the program with the `#include` preprocessor directive or the `INCLUDE` statement. In addition, you can define a specific location of include files with the compiler options, `-I`*`dir`* and `-X`. See Searching for Include Files in Preprocessing.

Customizing Compilation Process

This section describes options that customize compilation process—preprocessing, compiling, and linking. In addition, it discusses various compilation output and debug options and also shows how little-endian-to-big-endian conversions are enabled for unformatted sequential files.

You can find information on the link-time libraries used by compiler, compiler diagnostics, and mixing C and Fortran in the corresponding sections.

Specifying Alternate Tools and Locations

The Intel® Fortran Compiler lets you specify alternate to default tools and locations for preprocessing, compilation, assembly, and linking. Further, you can invoke options specific to your alternate tools on the command line. This functionality is provided by `-Qlocation` and `-Qoption`.

Specifying an Alternate Component (`-Qlocation,`*`tool`*`,`*`path`*)

`-Qlocation` enables to specify the pathname locations of supporting tools such as the assembler, linker, preprocessor, and compiler. This option's syntax is:

`-Qlocation,`*`tool`*`,`*`path`*

`tool`	Designates one or more of these tools: `fpp` Intel Fortran preprocessor `f` Fortran compiler (`f90com`) `asm` IA-32 assembler `ias` Itanium® assembler `link` Linker (`ld(1)`)
`path`	The location of the component.

Example:

`prompt``>`**`ifc -Qlocation,`*`fpp,/usr/preproc myprog.f`**

Passing Options to Other Tools (`-Qoption,`*`tool`*`,`*`opts`*)

`-Qoption` passes an option specified by *`opts`* to *`tool`*, where *`opts`* is a comma-separated list of options. The syntax for this option is:

`-Qoption,`*`tool`*`,`*`opts`*

tool	Designates one or more of these tools: `fpp` Intel Fortran preprocessor `f` Fortran compiler (`f90com`) `link` Linker (`ld(1)`)
opts	Indicates one or more valid argument strings for the designated program.

If the argument contains a space or tab character, you must enclose the entire argument in quotation characters (" "). You must separate multiple arguments with commas including those in quotation marks.

The following example directs the linker to link with alternate I/O library for mixed output with the C language for respective targeted compilations.

IA-32 applications:

prompt>**ifc -Qoption,*link,-C90 prog1.f***

Itanium®-based applications:

prompt>**efc -Qoption,*link,-C90 prog1.f***

Preprocessing

This section describes the options you can use to direct the operations of the preprocessor. Preprocessing performs such tasks as macro substitution, conditional compilation, and file inclusion. You can use the preprocessing options to direct the operations of the preprocessor from the command line. The compiler preprocesses files as an optional first phase of the compilation.

The Intel® Fortran Compiler provides the `fpp` binary to enable preprocessing. If you want to use another preprocessor, you must invoke it before you invoke the compiler. Source files that use a `.fpp` or `.F` file extension are automatically preprocessed.

> **Caution**
> Using a preprocessor that does not support Fortran can damage your Fortran code, especially with `FORMAT` statements. For example, `FORMAT (\\I4)` changes the meaning of the program because the backslash "`\`" indicates end-of-record.

Preprocessor Options

Use the options in this section to control preprocessing from the command line. If you specify neither option, the preprocessed source files are not saved but are passed directly to the compiler. Table that follows provides a summary of the available preprocessing

options.

Option	Description
-A[-]	Removes all predefined macros.
-D*name*=*{#|text}]*	Defines the macro name and associates it with the specified value. The default (-D*name*) defines a macro with value =1.
-E	Directs the preprocessor to expand your source module and write the result to standard output.
-EP	Same as -E but does not include #line directives in the output.
-F	Preprocess to an indicated file.

-fpp{n}	Uses the fpp preprocessor on Fortran source files.
	n=0: disable CVF and #directives n=1: enable CVF conditional compilation and #directives (default)
	n=2: enable only #directives,
	n=3: enable only CVF conditional compilation directives.

-P	Directs the preprocessor to expand your source module and store the result in a file in the current directory.
-U*name*	Eliminates any definition currently in effect for the specified macro.
-I*dir*	Adds directory to the include file search path.
-X	Removes standard directories from the include file search path.

Preprocessing Fortran Files

You do not usually preprocess Fortran source programs. If, however, you choose to preprocess your source programs, you must use the preprocessor fpp, or the preprocessing capability of a Fortran compiler. It is recommended to use fpp, which is the preprocessor supplied with the Intel® Fortran Compiler.

The compiler driver automatically invokes the preprocessor, depending on the source filename suffix and the option specified. For example, to preprocess a source file that contains standard Fortran preprocessor directives, then pass the preprocessed file to the compiler and linker, enter the following command:

IA-32 applications:

prompt>**ifc *source.fpp/source.F90***

Itanium®-based applications:

```
prompt>efc source.fpp/source.F90
```

The `.fpp` or `.F90` file extension invokes the preprocessor. Note the capital `F` in the file extension to produce the effect.

Note
Using the preprocessor can make debugging difficult. To get around this, you can save the preprocessed file (`-P`), and compile it separately, so that the proper file information is recorded for the debugger.

Enabling Preprocessing with CVF

You can enable the Preprocessor for any Fortran file by specifying the `-fpp` option. With `-fpp`, the compiler automatically invokes the `fpp` (preprocessor) to preprocess files with the `.f`, `.ftn`, `.for` or `.f90` extension in the mode set by `n`:

n=0: disable CVF and `#directives`

n=1: enable CVF conditional compilation and `#directives`; `-fpp1` is the default when the preprocessor is invoked.

n=2: enable only `#directives`

n=3: enable only CVF conditional compilation directives.

Note
Option `-openmp` automatically invokes the preprocessor.

String Constants for IA-32 Systems

Intel Fortran `fpp` conforms to `cpp` and accepts the `cpp` style directives. `cpp` prohibits the use of a string constant value in `#if` expression. So `fpp` won't support it either.

```
#define system "ia32"
#if system == "ia32"
void main() {
printf("ia32\n");
}
#else
int main() {
printf("non ia32\n");
}#endif
```

Preprocessing Only: `-E`, `-EP`, `-F`, and `-P`

Use either the `-E`, `-P`, or the `-F` option to preprocess your `.fpp` source files without compiling them.

When you specify the `-E` option, the Intel® Fortran Compiler's preprocessor expands your source file and writes the result to standard output. The preprocessed source contains `#line` directives, which the compiler uses to determine the source file and line number during its next pass. For example, to preprocess two source files and write them to stdout, enter the following command:

IA-32 applications:

prompt>**ifc -E *prog1.fpp prog2.fpp***

Itanium®-based applications:

prompt>**efc -E *prog1.fpp prog2.fpp***

When you specify the `-P` option, the preprocessor expands your source file and stores the result in a file in the current directory. By default, the preprocessor uses the name of each source file with the `.f` extension, and there is no way to change the default name. For example, the following command creates two files named `prog1.f` and `prog2.f`, which you can use as input to another compilation:

IA-32 applications:

prompt>**ifc -P *prog1.fpp prog2.fpp***

Itanium-based applications:

prompt>**efc -P *prog1.fpp prog2.fpp***

The `-EP` option can be used in combination with `-E` or `-P`. It directs the preprocessor to not include `#line` directives in the output. Specifying `-EP` alone is the same as specifying `-E` and `-EP`.

> **Caution**
> When you use the `-P` option, any existing files with the same name and extension are not overwritten and the system returns the error message invalid preprocessor output file.

Searching for Include and `.mod` Files

Include files are brought into the program with the `#include` preprocessor directive or the `INCLUDE` statement. To locate such included files, the compiler searches by default for the standard include files in the directories specified in the `INCLUDE` environment variable. In addition, you can specify the compiler options, `-I` and `-X`.

Specifying and Removing Include Directory Search: `-I, -X`

You can use the `-I` option to indicate the location of include files and .mod files. To prevent

the compiler from searching the default path specified by the INCLUDE environment variable, use -X option.

You can specify these options in the configuration files, ifc.cfg for IA-32 or efc.cfg for Itanium®-based applications or on the command line.

Specifying an Include Directory, -I*dir*

Included files are brought into the program with a #include preprocessor directive or a Fortran INCLUDE statement. Use the -I*dir* option to specify an alternative directory to search for include files.

Files included by the Fortran INCLUDE statement are normally referenced in the same directory as the file being compiled. The -I option may be used more than once to extend the search for an INCLUDE file into other directories.

Directories are searched for include files in this order:

- directory of the source file that contains the include
- directories specified by the -I option
- current working directory
- directories specified with the INCLUDE environment variable

Compiling an Input File from a Different Directory

If you need to compile an input file that resides in a directory other than default (that is, the directory where you issue a compilation command) and if your code contains an INCLUDE statement, you must use the -I*dir* option on your command line. For example:

IA-32 applications:

prompt>**ifc -I*dir* *dir/file.f90***

Itanium®-based applications:

prompt>**efc -I*dir* *dir/file.f90***

where *dir* is the directory path where the file, *file.f90* , you need to compile resides.

Specifying the .mod Files Directory

The programs that require modules located in multiple directories can be compiled using the -I*dir* option to locate the .mod files (modules) that should be included in the program. For specifying the directory to locate .mod files, see Searching and Locating the .mod Files in Large-Scale Projects.

Removing Include Directories, -X

Use the -X option to prevent the compiler from searching the default path specified by the INCLUDE environment variable.

You can use the -X option with the -I option to prevent the compiler from searching the default path for include files and direct it to use an alternate path. For example, to direct the compiler to search the path /alt/include instead of the default path, do the following:

IA-32 applications:

prompt>**ifc -X -I*/alt/include newmain.f***

Itanium-based applications:

prompt>**efc -X -I*/alt/include* n*ewmain.f***

Defining Macros

You can use the /D option to define the assertion and macro names to be used during preprocessing. The -U*name* option disable macros.

Use the -D option to define a macro. This option performs the same function as the #define preprocessor directive. The format of this option is:

-D*name[=value(*text*]*

where

name	The name of the macro to define.
*value [=*text*]*	Indicates a value to be substituted for name.

If you do not enter a *value*, *name* is set to 1. The *value* should be enclosed in the quotation marks if it contains spaces or special characters.

Preprocessing replaces every occurrence of *name* with the specified *value*. For example, to define a macro called *SIZE* with the *value* 100 use the following command:

IA-32 applications: *prompt*>ifc -D*SIZE*=100 *prog1.f*

Itanium®-based applications: *prompt*>efc -D*SIZE*=100 *prog1.f*

Preprocessing replaces all occurrences of *SIZE* with the specified value before passing the preprocessed source code to the compiler. Suppose the program contains the declaration:

```
REAL   VECTOR(SIZE)
```

In the code sent to the compiler, the value 100 replaces *SIZE* in this declaration, and in every other occurrence of the name *SIZE*.

Predefined Macros

The predefined macros available for the Intel® Fortran Compiler are described in the table below. The **Default** column describes whether the macro is enabled (ON) or disabled (OFF) by default. The **Disable** column lists the option which disables the macro.

Macro Name	Default	Architecture	Description - When Used
`__EFC`	ON	Itanium architecture	Identifies the Intel Fortran Compiler
`__IFC`	ON	IA-32	Identifies the Intel Fortran Compiler
`__linux__`	ON	IA-32	Defined for Linux* applications
`_M_IA64_linux`	ON	Itanium® architecture	Defined for Itanium-based Linux applications
`_M_IX86=`*n*	ON,*n*=700	IA-32	Defined based on the processor option you specify: *n*=500 if you specify `-tpp5` *n*=600 if you specify `-tpp6` *n*=700 if you specify `-tpp7`
`_PGO_INSTRUMENT`	OFF	Both	Defined when you compile with `-prof_gen` or `-prof_genx` options.

Suppressing Macros

The `-U` option directs the preprocessor to suppress an automatic definition of a macro. Use the `-U`*name* option to suppress any macro definition currently in effect for the specified *name*. The `-U` option performs the same function as an `#undef` preprocessor directive.

Preprocessor Macro for OpenMP*

A preprocessor macro is defined which may be useful for running OpenMP* depending on the compiler environment:

`_OPENMP`

This macro has the form YYYYMM where YYYY is the year and MM is the month of the OpenMP Fortran specification supported.

Compilation

This section describes all the Intel® Fortran Compiler options that determine the compilation and linking process and their output. By default, the compiler converts source code directly to an executable file. Appropriate options enable you to control the process and obtain desired output file produced by the compiler.

Having control of the compilation process means, for example, that you can create a file at any of the compilation phases such as assembly, object, or executable with `-P` or `-c` options. Or you can name the output file or designate a set of options that are passed to the linker with the `-S`, `-o` options. If you specify a phase-limiting option, the compiler produces a separate output file representing the output of the last phase that completes for each primary input file.

You can use the command line options to display and check for certain aspects of the compiler's behavior. You can use these options to see which options and files are passed by the compiler driver to the component executables `f90com` and `ld(1)` (option `-sox[-]`).

Linking is the last phase in the compilation process discussed in a separate section. See the Linking options.

A group of options monitors the outcome of Intel compiler-generated code without interfering with the way your program runs. These options control some computation aspects, such as allocating the stack memory, setting or modifying variable settings, and defining the use of some registers.

The options in this section provide you with the following capabilities:

- GCC* compatibility
- controlling compilation
- monitoring data settings
- specifying the output files or directories

Finally, the output options are summarized in Compiler Output Options Summary.

Controlling Compilation

You can control and modify the compilation process with the option sets as follows.

Controlling Compilation Phases

You can control which compilation phases you need to include in the compilation process.

- The `-c` option directs the compiler to compile, assemble and generate object file(s), but do not link.

- The `-S` option stops compiler at generating assembly files.
- If you need to link additional files and/or libraries, you use the `-l`*name* option. For example, if you want to link `libm.a`, the command is:

IA-32 compiler:

prompt>**ifc *a.f* -l*m***

Itanium® compiler:

prompt>**efc *a.f* -l*m***

Aliasing

The following options manage compiler aliasing:

- `-falias` assumes aliasing in a program
- `-fno-alias` assumes no aliasing in a program
- `-ffnalias` assumes aliasing within functions
- `-fno-fnalias` assumes no aliasing within functions, but assumes aliasing across calls

Translating Other Code to Fortran

The `/Tf`*file* option enables you to treat a text file as if it contains Fortran code. This option is used if you have a Fortran file that has other than the `.f`/`.for`/`.f90` extension or no extension, and you need to compile it.

For example:

prompt>**ifc -Tf*a.f95 b.f***

The above command will compile both `a.f95` and `b.f` files as Fortran, link them, and create executable `a`.

Profiling Support

Profiling information identifies those parts of your program where improving source code efficiency would most likely improve runtime performance.

The options supporting profiling are `-p` and `-qp`, and `-pg`. (`-pg` is used for IA-32 only)

`-p` and `-qp` set up profiling by periodically sampling the value of the program counter for

use with the postprocessor `prof` tool.

These options only affect loading. When loading occurs, these options replace the standard runtime startup routine option with the profiling runtime startup routine. When profiling occurs, an output file is produced, which contains execution-profiling data for use with the postprocessor `prof` command.

`-pg` (IA-32 only) sets up profiling for `gprof` tool, which produces a call graph showing the execution of the program. When programs are linked with the `-pg` option and then run, these files produced:

- a file containing a dynamic call graph and profile.
- a file containing a summarized dynamic call graph and profile.

To display the output, run `gprof` on the file containing a dynamic call graph and profile.

Saving Compiler Version and Options Information, `-sox[-]`

You can save the compiler version and options information in the executable with `-sox`. The size of the executable on disk is increased slightly by the inclusion of these information strings. The default is `-sox-`.

The `-sox` option forces the compiler to embed in each object file a string that contains information on the compiler version and compilation options for each source file that has been compiled. When you link the object files into an executable file, the linker places each of the information strings into the header of the executable. It is then possible to use a tool, such as a strings utility, to determine what options were used to build the executable file.

> **Note**
> For Itanium®-based applications, the `-sox` option is accepted for compatibility, but it does not have any effect.

Monitoring Data Settings

The options described below provide monitoring the outcome of Intel compiler-generated code without interfering with the way your program runs.

Specifying Structure Tag Alignments

Use the `-Zp{n}` option to determine the alignment constraint for structure declarations, on n-byte boundary (n = 1, 2, 4, 8, 16). Generally, smaller constraints result in smaller data sections while larger constraints support faster execution.

For example, to specify 2 bytes as the alignment constraint for all structures and unions in the file `prog1.f`, use the following command:

IA-32 systems: *prompt*>**ifc -Zp2 *prog1.f***

The default for IA-32 systems is `-Zp4`.

Itanium®-based systems: *prompt*>**efc -Zp2 *prog1.f***
The default for Itanium-based systems is `-Zp8`.

The `-Zp16` option enables you to align Fortran structures such as common blocks. For Fortran structures, see STRUCTURE statement in Chapter 10 of *Intel® Fortran Programmer's Language Reference Manual*.

The `-align` option applies mainly to structures and analyzes and reorders memory layout for variables and arrays and basically functions as `-Zp{n}`. You can disable either option with
`-noalign`.

The `-pad` option is effectively not different from `-align` when applied to structures and derived types. However, the scope of `-pad` is greater because it applies also to common blocks, derived types, sequence types, and Vax structures.

Allocation of Zero-initialized Variables, `-nobss_init`

By default, variables explicitly initialized with zeros are placed in the BSS section. But using the
`-nobss_init` option, you can place any variables that are explicitly initialized with zeros in the DATA section if required.

Correcting Computations for IA-32 Processors, `-0f_check` (IA-32 Systems)

Specify the `-0f_check` option to avoid the incorrect decoding of the instructions that have 2-byte opcodes with the first byte containing `0f`. In rare cases, the Pentium® processor can decode these instructions incorrectly.

The ebp Register Usage (IA-32 Systems)

The `-fp` option disables the use of the `ebp` register in optimizations. The option directs to use the `ebp`-based stack frame for all functions. For details on the correlation between the `ebp` register use for optimizations and debugging, see -fp Option and Debugging.
The `-fp` option is disabled by default or when `-O1` or `-O2` (see optimization-level options) are specified.

Flushing to Zero Denormal Values, `-ftz` (Itanium®-based Systems)

Option `-ftz` flushes denormal results to zero when the application is in the gradual underflow mode. Use this option if the denormal values are not critical to application behavior.

Flushing the denormal values to zero with `-ftz` may improve performance of your application.

The default status of `-ftz` is OFF. By default, the compiler lets results gradually underflow.

Little-endian-to-Big-endian Conversion (IA-32)

The little-endian-to-big-endian conversion feature is intended for Fortran unformatted input/output operations. It enables the development and processing of files with big-endian data organization on the IA-32-based processors, which usually process the data in the little endian format.

The feature also enables processing of the files developed on processors that accept big-endian data format and producing the files for such processors on IA-32-based little-endian systems.

The little-endian-to-big-endian conversion is accomplished by the following operations:

- The `WRITE` operation converts little endian format to big endian format.
- The `READ` operation converts big endian format to little endian format.

The feature enables the conversion of variables and arrays (or array subscripts) of basic data types. Derived data types are not supported.

Little-to-Big Endian Conversion Environment Variable

In order to use the little-endian-to-big-endian conversion feature, specify the numbers of the units to be used for conversion purposes by setting the `F_UFMTENDIAN` environment variable. Then, the `READ`/`WRITE` statements that use these unit numbers, will perform relevant conversions. Other `READ`/`WRITE` statements will work in the usual way.

In the general case, the variable consists of two parts divided by a semicolon. No spaces are allowed inside the `F_UFMTENDIAN` value. The variable has the following syntax:

```
F_UFMTENDIAN=MODE | [MODE;] EXCEPTION
```

where:

```
MODE = big | little
EXCEPTION = big:ULIST | little:ULIST | ULIST
ULIST = U | ULIST,U
U = decimal | decimal -decimal
```

- `MODE` defines current format of data, represented in the files; it can be omitted.
  The keyword `little` means that the data have little endian format and will not be converted. For IA-32 systems, this keyword is a default.
  The keyword `big` means that the data have big endian format and will be converted.
  This keyword may be omitted together with the colon.

- EXCEPTION is intended to define the list of exclusions for MODE; it can be omitted. EXCEPTION keyword (little or big) defines data format in the files that are connected to the units from the EXCEPTION list. This value overrides MODE value for the units listed.

- Each list member U is a simple unit number or a number of units. The number of list members is limited to 64.
  decimal is a non-negative decimal number less than 2^{32}.

Converted data should have basic data types, or arrays of basic data types. Derived data types are disabled.

Command lines for variable setting with different shells:

Sh: **export F_UFMTENDIAN=MODE;EXCEPTION**

Csh: **setenv F_UFMTENDIAN MODE;EXCEPTION**

 **Note**

Environment variable value should be enclosed in quotes if semicolon is present.

Another Possible Environment Variable Setting

The environment variable can also have the following syntax:

F_UFMTENDIAN=u[,u] . . .

Command lines for the variable setting with different shells:

- Sh: **export F_UFMTENDIAN=u[,u] . . .**

- Csh: **setenv F_UFMTENDIAN u[,u] . . .**

See error messages that may be issued during the little endian – big endian conversion. They are all fatal. You should contact Intel if such errors occur.

Little-to-Big Endian Conversion Usage Examples

1. F_UFMTENDIAN=big

   All input/output operations perform conversion from big-endian to little-endian on READ and from little-endian to big-endian on WRITE.

2. F_UFMTENDIAN="little;big:10,20"
or F_UFMTENDIAN=big:10,20
or F_UFMTENDIAN=10,20

In this case, only on unit numbers 10 and 20 the input/output operations perform big - little endian conversion.

3. `F_UFMTENDIAN="big;little:8"`

   In this case, on unit number 8 no conversion operation occurs. On all other units, the input/output operations perform big-little endian conversion.

4. `F_UFMTENDIAN=10-20`

   Define 10, 11, 12 … 19, 20 units for conversion purposes; on these units, the input/output operations perform big-little endian conversion.

5. Assume you set `F_UFMTENDIAN=10,100` and run the following program.

```
integer*4   cc4
integer*8   cc8
integer*4   c4
integer*8   c8
c4 = 456
c8 = 789

C  prepare a little endian representation of
data

open(11,file='lit.tmp',form='unformatted')
write(11) c8
write(11) c4
close(11)

C  prepare a big endian representation of data

open(10,file='big.tmp',form='unformatted')
write(10) c8
write(10) c4
close(10)

C  read big endian data and operate with them
on
C  little endian machine.

open(100,file='big.tmp',form='unformatted')
read(100) cc8
read(100) cc4

C  Any operation with data, which have been
read

C    . . .
close(100)
```

```
stop
end
```

Now compare `lit.tmp` and `big.tmp` files with the help of `od` utility.

```
> od -t x4 lit.tmp

0000000 00000008 00000315 00000000 00000008
0000020 00000004 000001c8 00000004
0000034

> od -t x4 big.tmp

0000000 08000000 00000000 15030000 08000000
0000020 04000000 c8010000 04000000
0000034
```

You can see that the byte order is different in these files.

Specifying Compilation Output

When compiling and linking a set of source files, you can use the `-o` or `-S` option to give the resulting file a name other than that of the first source or object file on the command line.

`-c`	Compile to object only (`.o`), do not link.
`-S`	Produce assembly file or directory for multiple assembly files. The compilation stops at producing the assembly file.
`-o`*`file`*	Produce an output file based on the phase options used previously: none, `-c` or `-S`. If no phase option has been used, produces an executable and places it in specified *`file`*. Combined with `-S`, indicates assembly file or directory for multiple assembly files. Combined with `-c`, indicates object file name or directory for multiple object files.

If you are processing a single file, you can use the `-o`*`file`* option to specify an alternate name for an object file (`.o`), an assembly file (`.s`) or an executable file. You can also use these options to override the default filename extensions: `.o` and `.s`.

See Compilation Output options summary.

Default Output Files

The default command line does not include any options and has a Fortran source file as its input argument:

IA-32 compiler:

prompt>**ifc *a.f90***

Itanium® compiler:

prompt>**efc *a.f90***

The default compiler command produces an `a.out` executable file. If the `-c` option was used, the compiler command also produces an object file, `a.o`, and places it in the current directory.

You can compile more than one input files:

IA-32 compiler:

prompt>**ifc *x.f90 y.f90 z.f90***

Itanium compiler:

prompt>**efc *x.f90 y.f90 z.f90***

The above command will do the following:

- compile and link three input source files
- produce three object files and assign the names of the respective source files: `x.o`, `y.o`, and `z.o`
- produce an executable file and assign to it the default name `a.out`
- place all the files in the current directory.

To generate assembly files, use the `-S` option. The compilation stops at producing the assembly file.

Specifying Executable Files

You can use the `-o`*`file`* option to specify an alternate name for an executable file. This is especially useful when compiling and linking a set of input files. You can use the `-o`*`file`* option to give the resulting file a name other than that of the first input file (source or object) on the command line.

In the next example, the command produces an executable file named `outfile` as a result of compiling and linking two source files.

IA-32 compiler:

```
prompt>ifc -ooutfile file1.f90 file2.f90
```

Itanium® compiler:

```
prompt>efc -ooutfile file1.f90 file2.f90
```

Without the `-o`*`outfile`* option, the command above produces an executable file named a.out, the default executable file name.

Specifying Object Files

The compiler command always generates and keeps object files of the input source files and by default places them in the current directory. You can use the `-o`*`file`* options to specify an alternate name for an object file.

For example:

IA-32 compiler:

```
prompt>ifc -ofile.o x.f90
```

Itanium® compiler:

```
prompt>efc -ofile.o x.f90
```

In the above example, `-o` assigns the name `file.o` to an output object file rather than the default `x.o`.

To generate object files, specify a different object file name, and suppress linking, use `-c` and `-o` combination.

IA-32 applications:

```
prompt>ifc -c -ofile.o x.f90
```

Itanium compiler:

```
prompt>efc -c -ofile.o x.f90
```

`-o` assigns the name `file.o` to an output object file rather than the default (`x.o`)

`-c` directs the compiler to suppress linking.

Specifying Assembly Files

You can use the `-S` option to generate an assembly file. The compilation stops at producing the assembly file. To specify an alternate name for this assembly file, use the `-o`*`file`*

option .

IA-32 compiler:

```
prompt>ifc -S -ofile.s x.f90
```

Itanium® compiler:

```
prompt>efc -S -ofile.s x.f90
```

In the above example, -S tells the compiler to generate an assembly file, while **-o*file.s*** assigns to it the name `file.s` rather than the default `x.s`.

The option `-S` tells compiler to:

- generate an assembly file of the source file
- use the name of the source file as a default assembly output file name
- place this file in the current directory.

Note

The `-S` option stops the compiler upon generating and saving the assembly files. Without the `-S` option, the compiler proceeds to generating object files without saving the assembly files.

Producing Assembly Files with Annotations and Comments

Options `-fcode-asm` and `-fsource-asm` produce annotations in assembly files as follows:

- `-fcode-asm` and inserts code byte information in the assembly file
- `-fsource-asm` and inserts high-level source code in the assembly file

In addition, the options `-fverbose-asm` and `-fnoverbose-asm` enable and disable, respectively, inserting comments containing compiler version and options used in the assembly file. The `-fverbose-asm` option is enabled by default when producing an assembly file with `-fcode-asm` or `-fsource-asm`.

Compiler Output Options Summary

If no errors occur during processing, you can use the output files from a particular phase as input to a later compiler invocation. The executable file is produced when you do not specify any phase-limiting option. The filename of the first source or object file specified with an absent suffix, is the default for the executable object file from the linker.

The table below describes the options to control the output.

Last Phase Completed	Option	Compiler Input	Compiler Output
preprocessing	`-P`, `-E`, or `-EP`	source files	preprocessed files, see Preprocessing
compile only	`-c`	source	Compile to object only (`.o`), do not link.
assembly only	`-S`	source	Compile to assembly file only (`.s`) and stop.
compilation, linking, or assembly	`-o,`*`name`* `-o,`*`name`*	source, assembly, or object files	Assigns a name of your choice to an output file
syntax checking	`-y`	source files preprocessed files	diagnostic list
linking	(default)	source files preprocessed files assembly files object files libraries	executable file, map file

Using the Assembler to Produce Object Code

By default the compiler generates an object file directly without going through the assembler. But if you want to link some specific input file to the Fortran project object file, you can use the `-use_asm` option to tell the compiler to use the Linux* Assembler for IA-32 systems or Itanium® Assembler for Itanium®-based systems.

prompt>**ifc -use_asm *file1.f***

prompt>**efc -use_asm *file1.f***

The above command generates an `file1.o` object file which you can link with the Fortran object file(s) of the whole project.

Listing Options

The following options produce a source listing to the standard output, which by default is the screen.

- The `-list` option writes a listing of the source file to standard output (typically, your terminal screen), including any error or warning messages. The errors and warnings are

also output to standard error, `stderr`.

- The `-list -show`*`include`* prints a source listing to `stdout` with contents of `include` files expanded.

Linking

This topic describes the options that enable you to control and customize the linking with tools and libraries and define the output of the linking process. See the summary of linking options.

Note
These options are specified at compile time and have effect at the linking time.

Options to Link to Tools and Libraries

The following options enable you to link to various tools and libraries:

`-Bdynamic`	Used with `-l`*`name`* (see below), enables dynamic linking of libraries at run time. Compared to static linking, results in smaller executables.
`-Bstatic`	Enables linking a user's library statically.
`-C90`	Link with alternate I-O library for mixed output with the C language.
`-i_dynamic`	Enables to link the shared object versions of the Intel-provided libraries dynamically.
`-l`*`name`*	Link with a library indicated in name. For example, `-l`*`m`* indicates to link with the math library.
`-L`*`dir`*	Instructs linker to search *`dir`* for libraries.
`-posixlib`	Enables or disable linking with POSIX* library.
`-shared`	Instructs the compiler to build the Dynamic Shared Object (DSO) instead of an executable.
`-static`	Enables to link shared libraries (`.so`) statically at compile time. Compared to dynamic linking, results in larger executables. When `-static` is not used: • `/lib/ld-linux.so.2` is linked in • `libm`, `libcxa`, and `libc` are linked dynamically • all other libraries are linked statically When `-static` is used:

	• `/lib/ld-linux.sl.2` is not linked in • all other libraries are linked statically
`-Vaxlib`	Enable or disable linking with portability library.

Controlling Linking and its Output

`-L`*`dir`*	Instruct linker to search for *`dir`* libraries.

See Libraries for more information on using them.

Suppressing Linking

Use the `-c` option to suppress linking. Entering the following command produces the object files `file.o` and `file2.o`, but does not link these files to produce an executable file.

IA-32 compiler:

prompt>**ifc -c *file.f file2.f***

Itanium® compiler:

prompt>**efc -c *file.f file2.f***

Note
The preceding command does not link these files to produce an executable file.

Debugging

This section describes the basic command line options that you can use as tools to debug your compilation and to display and check compilation errors. The options in this section enable you to:

- support for symbolic debugging
- compile only designated lines and debug statements
- check the source files for syntax errors before creating output file

Support for Symbolic Debugging

Use the `-g` option to direct the compiler to generate code to support symbolic debugging. For example:

IA-32 applications: *prompt*>ifc -g *prog1.f*

Itanium®-based applications: *prompt*>efc -g *prog1.f*

The compiler lets you generate code to support symbolic debugging while the `-O1`, or `-O2` optimization options are specified on the command line along with `-g`.

If you specify the `-O1`, or `-O2` options with the `-g` option, you can receive these results:

- some of the debugging information returned may be inaccurate as a side-effect of optimization.
- for IA-32 applications, `-O1`, or `-O2` options disable the `-fp` option. See -fp Option and Debugging.

Debugging and Assembling

The compiler does not support the generation of debugging information in assembly files. If you specify the `-g` option with `-S`, the assembly listing file is generated without debugging information, but if you further produce an object file, it will contain debugging information. If you link the object file and then use the GDB debugger on it, you will get full symbolic representation.

Compiling Source Lines with Debugging Statements, -DD

This option is useful for the inclusion or exclusion of debugging lines. Use the `-DD` option to compile source lines containing user debugging statements.

The -DD Option

Debugging statements included in a Fortran program source are indicated by the letter D in column 1. The `-DD` option instructs the compiler to treat a `D` in column 1 of Fortran source as a space character. The rest of that line is then parsed as a normal Fortran statement.

For example, to compile any debugging statements in program `prog1.f`, enter the following command:

prompt>**ifc -DD *prog1.f***

The above command causes the debugging statement

```
D      PRINT *, "I= ",I
```

embedded in the `prog1.f` to execute and print lines designated for debugging.

By default, the compiler takes no action on these statements. In the following example, if `-DD` is not specified (default), the `D` line is ignored:

```
do 10 i = 1, n
     a(i) = b(i)
D    write (*,*) a(i)
10 continue
```

But when `-DD` is specified, the compiler sees a `write` statement as if the code is:

```
   do 10 i = 1, n
     a(i) = b(i)
     write (*,*) a(i)
10 continue
```

The `-DX` and `-DY` Options

Two additional distinctions to compile source lines containing user debugging statements are also available with these variations of the `-DD` option:

- `-DX` compiles debug statements indicated by a `X` (not an `x`) in column 1; if this option is not set these lines are treated as comments.
- `-DY` compiles debug statements indicated by a `Y` (not an `y`) in column 1; if this option is not set these lines are treated as comments.

Parsing for Syntax Only

Use the `-y` or `-syntax` option to stop processing source files after they have been parsed for Fortran language errors. This option gives you a way to check quickly whether sources are syntactically and semantically correct. The compiler creates no output file. In the following example, the compiler checks a file named `prog1.f`. Any diagnostics appear on the standard error output and in a listing, if you have requested one.

IA-32 applications: *prompt*`>ifc -y` ***`prog1.f`***

Itanium®-based applications: *prompt*`>efc -y` ***`prog1.f`***

Debugging and Optimizations

It is best to make your optimization and/or debugging choices explicit:

- If you need to debug your program excluding any optimization effect, use the `-O0` option, which turns off all the optimizations.
- If you need to debug while still use optimizations, you can specify the `-O1` or `-O2` options on the command line along with `-g`.

If you do not make your optimization choice explicit when `-g` is specified, the `-g` option implicitly disables optimization (as if `-O0` were specified).

-fp Option and Debugging (IA-32 only)

The -fp option disables use of the ebp register in optimizations, and can result in slightly less efficient code. With this option, the compiler generates code for IA-32-targeted compilations without turning off optimization, so that a debugger can still produce a stack backtrace.

If you specify the -O1 or -O2 options, the -fp option is disabled. If you specify the -O0 option, -fp is enabled. Remember that the -fp option affects IA-32 applications only.

Summary

Refer to the table below for the summary of the effects of using the -g option with the optimization options.

These options	Imply these results
-g	debugging information produced, -O0 enabled, -fp enabled for IA-32-targeted compilations.
-g -O1	debugging information produced, -O1 optimizations enabled, -fp disabled for IA-32-targeted compilations
-g -O2	debugging information produced, -O2 optimizations enabled, -fp disabled for IA-32-targeted compilations
-g -O3 -fp	debugging information produced, -O3 optimizations enabled, -fp enabled for IA-32-targeted compilations.
-g -ip	limited debugging information produced, -ip option enabled.

Fortran Language Options

The Intel® Fortran Compiler implements Fortran language-specific options, which enable you to set or specify:

- set data types and sizes
- define source program characteristics
- set arguments and variables
- allocate common blocks

For the size or number of Fortran entities the Intel® Fortran Compiler can process, see Maximum Size and Number table.

Setting Integer and Floating-point Data Types

See the summary of these options.

Integer Data

The `-i2`, `-i4`, and `-i8` options specify that all quantities of `INTEGER` type and unspecified `KIND` occupy two, four or eight bytes, respectively. All quantities of `LOGICAL` type and unspecified `KIND` also occupy two, four or eight bytes, respectively.

All logical constants and all small integer constants occupy two, four or eight bytes, respectively.

The default is four bytes, `-i4`.

Floating-point Data

The `-r{4|8|16}` option defines the `KIND` for real variables in 4, 8, and 16 bytes. The default is `-r4`.

The `-r8`, `-autodouble`, and `-r16` options specify floating-point data.

The `-r8` option directs the compiler to treat all variables, constants, functions and intrinsics as `DOUBLE PRECISION`, and all complex quantities as `DOUBLE COMPLEX`. The `-autodouble` option has the same effect as the `-r8` option.

The `-r16` option directs the compiler to treat all variables, constants, functions and intrinsics as `DOUBLE PRECISION`, and all complex quantities as `DOUBLE COMPLEX`. This option changes the default size of real numbers to 16 bytes.

Source Program Features

The options that enable the compiler to process a source program in a beneficial way for or required by the application, can be divided in two groups described in the two sections below. See a summary of these options.

Program Structure and Format

DO loops

The `-onetrip` option directs the compiler to compile `DO` loops at least once. By default Fortran `DO` loops are not performed at all if the upper limit is smaller than the lower limit. The option `-1` has the same effect. This supports old programs from the Fortran–66 standard, when all `DO` loops executed at least once.

Fixed Format Source

The `-FI` option specifies that all the source code is in fixed format; this is the default except for files ending with the extension `.f`, `.for`, `.ftn`.

`-132` permits fixed form source lines to contain up to 132 characters. The `-extend_source`, option has the same effect as `-132`.

Free Format Source

-FR options Specifies that all the source code is in Fortran free format; this is the default for files ending with the suffix `.f90`.

Character Definitions

The `-pad_source` option enforces the acknowledgment of blanks at the end of a line.

The `-us` option appends an underscore to external subroutine names. `-nus` disables appending an underscore to an external subroutine name.

The `-nus[`*`file`*`]` option directs to not append an underscore to subroutine names listed in *`file`*. Useful when linking with C routines.

The `-nbs` option directs the compiler to treat backslash (\) as a normal graphic character, not an escape character. This may be necessary when transferring programs from non-UNIX* environments, for example from VAX* VMS*. See Escape Characters.

Compatibility with Platforms and Compilers

This group discusses options that enable compatibility with other compilers.

Cross-platform

The `-ansi_alias[-]` enables (default) or disables assumption of the program's ANSI conformance. Provides cross-platform compatibility. This option is used to make assumptions about out-of-bound array references and pointer references. For gcc compatibility, the `-ansi_alias` option is accepted. The option is ON by default.

The option directs the compiler to assume the following:

- Arrays are not accessed out of arrays' bounds.
- Pointers are not cast to non-pointer types and vice-versa.
- References to objects of two different scalar types cannot alias. For example, an object of type `integer` cannot alias with an object of type `real` or an object of type `real` cannot alias with an object of type `double precision`.

If your program satisfies the above conditions, setting the `-ansi_alias` option will help the compiler better optimize the program. However, if your program may not satisfy one of the above conditions, the option must be disabled, as it can lead the compiler to generate incorrect code.

DEC* VMS

The `-dps`, option enables (default) or disables DEC* parameter statement recognition. Basically, the
`-dps` option determines how the compiler treats the alternate syntax for `PARAMETER` statements, which is:

```
PARAMETER par1=exp1 [, par2=exp2] ...
```

This form does not have parentheses around the assignment of the constant to the parameter name. With this form, the type of the parameter is determined by the type of the expression being assigned to it and not by any implicit typing.

By default, the compiler allows the alternate syntax for `PARAMETER` statements, `-dps`. To disable this form, specify `-nodps`.

The `-vms` option enables support for extensions to Fortran that were introduced by Digital* VMS Fortran compilers. The extensions are as follows:

- The compiler permits shortened, apostrophe-separated syntax for parameters in I/O statements. For example, a statement of the form: `WRITE(4'7) FOO` is permitted and is equivalent to `WRITE(UNIT=4, REC= 7) FOO`.
- The compiler assumes that the value specified for `RECL` in an `OPEN` statement is given in words rather than bytes. This option also implies `-dps`, even though `-dps` is on by default.

C Language

The `-lowercase` maps external routine names and symbol names (linker) to lowercase alphabetic characters. This option is useful when mixing Fortran with C programs.

The `-uppercase` maps external names to uppercase alphabetic characters.

Note
Do not use the `-uppercase` option in combination with `-Vaxlib` or `-posixlib`.

Escape Characters

For compatibility with C usage, the backslash (`\`) is normally used in Intel® Fortran Compiler as an escape character. It denotes that the following character in the string has a significance which is not normally associated with the character. The effect is to ignore the backslash character, and either substitute an alternative value for the following character or to interpret the character as a quoted value.

The escape characters recognized, and their effects, are described in the table below. Thus, `'ISN\'T'` is a valid string. The backslash (`\`) is not counted in the length of the string.

Escape Characters and Their Effect

Escape Character	Effect
`\n`	new line
`\t`	horizontal tab
`\v`	vertical tab
`\b`	backspace
`\f`	form feed
`\0`	null
`\'`	apostrophe (does not terminate a string)
`\"`	double quote (does not terminate a string)
`\\`	`\` (a single backslash)
`\x`	`x`, where `x` is any other character

Line Terminators

This information is useful for recent Linux* users after working with Windows*. The line terminators are different between Linux and Windows. On Windows, line terminators are `\r\n` while on Linux they are just `\n`. Typically, a file transfer program will take care of this issue for you if you transfer the file in text mode. If the file is transferred in binary mode (but the file is really text file), the problem will not be resolved by FTP.

Setting Arguments and Variables

These options can be divided into two major groups discussed below. See a summary of these options.

Automatic Allocation of Variables to Stacks

-auto

This option makes all local variables AUTOMATIC. Causes all variables to be allocated on the stack, rather than in local static storage. Variables defined in a procedure are otherwise allocated to the stack only if they appear in an AUTOMATIC statement, or if the procedure is recursive and the variables do not have the SAVE or ALLOCATABLE attributes. The option does not affect variables that appear in an EQUIVALENCE or SAVE statement, or those that are in COMMON. May provide a performance gain for your program, but if your program depends on variables having the same value as the last time the routine was invoked, your program may not function properly.

-auto_scalar

This option causes scalar variables of rank 0, except for variables of the COMPLEX or CHARACTER types, to be allocated on the stack, rather than in local static storage. Does not affect variables that appear in an EQUIVALENCE or SAVE statement, or those that are in COMMON. -auto_scalar may provide a performance gain for your program, but if your program depends on variables having the same value as the last time the routine was invoked, your program may not function properly. Variables that need to retain their values across subroutine calls should appear in a SAVE statement. This option is similar to -auto, which causes all local variables to be allocated on the stack. The difference is that -auto_scalar allocates only variables of rank 0 on the stack.

-auto_scalar enables the compiler to make better choices about which variables should be kept in registers during program execution. This option is on by default.

-save and -zero

Forces the allocation of all variables in static storage. If a routine is invoked more than once, this option forces the local variables to retain their values from the last invocation terminated. This may cause a performance degradation and may change the output of your program for floating-point values as it forces operations to be carried out in memory rather than in registers which in turn causes more frequent rounding of your results. Opposite of -auto. To disable -save, set -auto. Setting -save turns off both -auto and -auto-scalar.

The -zero option presets uninitialized variables to zero. It is most commonly used in conjunction with
-save.

Alignment, Aliases, Implicit None

Alignment

The `-align` option is a front-end option that changes alignment of variables in a `COMMON` block.

Example:

```
COMMON /BLOCK1/CH,DOUB,CH1,INT
INTEGER INT
CHARACTER(LEN=1) CH,CH1
DOUBLE PRECISION DOUB
END
```

The `-align` option enables padding inserted to assure alignment of `DOUB` and `INT` on natural alignment boundaries. The `-noalign` option disables padding.

Aliases

The `-common_args` option assumes that the "by-reference" subprogram arguments may have aliases of one another.

Implicit None

The `-u` and `-implicitnone` options set `IMPLICIT NONE` as the default.

Preventing CRAY* Pointer Aliasing

Option `-safe_cray_ptr` specifies that the CRAY* pointers do not alias with other variables. The default is OFF.

Consider the following example.

```
pointer (pb, b)
pb = getstorage()
do i = 1, n
b(i) = a(i) + 1
enddo
```

When `-safe_cray_ptr` is not specified (default), the compiler assumes that `b` and `a` are aliased. To prevent such an assumption, specify this option, and the compiler will treat `b(i)` and `a(i)` as independent of each other.

However, if the variables are intended to be aliased with CRAY pointers, using the `-safe_cray_ptr` option produces incorrect result. For the code example below, `-safe_cray_ptr` should not be used.

```
pb = loc(a(2))
do i=1, n
b(i) = a(i) +1
enddo
```

Allocating Common Blocks

The following two options are used for the common blocks:

`-Qdyncom"blk1,blk2 ..."`	Dynamically allocates `COMMON` blocks at runtime. See section Dynamic Common Option that follows.
`-Qloccom"blk1,blk2, ..."`	Enables local allocation of given `COMMON` blocks at run time. See Allocating Memory to Dynamic `COMMON` Blocks.

Dynamic Common Option

The `-Qdyncom` option dynamically allocates `COMMON` blocks at runtime. This option on the compiler command line designates a `COMMON` block to be dynamic, and the space for its data is allocated at runtime, rather than compile time. On entry to each routine containing a declaration of the dynamic `COMMON` block, a check is made of whether space for the `COMMON` block has been allocated. If the dynamic `COMMON` block is not yet allocated, space is allocated at the check time.

The following example of a command-line specifies the dynamic common option with the names of the `COMMON` blocks to be allocated dynamically at runtime:

IA-32 applications:

prompt>**ifc -Qdyncom"*BLK1,BLK2,BLK3*" *test.f***

Itanium®-based applications:

prompt>**efc -Qdyncom"*BLK1,BLK2,BLK3*" *test.f***

where `BLK1`, `BLK2`, and `BLK3` are the names of the `COMMON` blocks to be made dynamic.

Allocating Memory to Dynamic Common Blocks

The runtime library routine, `f90_dyncom`, performs memory allocation. The compiler calls this routine at the beginning of each routine in a program that contains a dynamic `COMMON` block. In turn, this library routine calls `_FTN _ALLOC()` to allocate memory. By default, the compiler passes the size in bytes of the `COMMON` block as declared in each routine to `f90_dyncom`, and then on to `_FTN_ALLOC()`. If you use the nonstandard extension having the `COMMON` block of the same name declared with different sizes in different routines, you may get a runtime error depending upon the order in which the routines containing the

COMMON block declarations are invoked.

The runtime library contains a default version of _FTN_ALLOC(), which simply allocates the requested number of bytes and returns.

Why Use a Dynamic Common

One of the primary reasons for using dynamic COMMON is to enable you to control the COMMON block allocation by supplying your own allocation routine. To use your own allocation routine, you should link it ahead of the runtime library routine. This routine must be written in the C language to generate the correct routine name.

The routine prototype is as follows:

```
void _FTN_ALLOC(void **mem, int *size, char *name);
```

where

mem	is the location of the base pointer of the COMMON block which must be set by the routine to point to the block memory allocated.
size	is the integer number of bytes of memory that the compiler has determined are necessary to allocate for the COMMON block as it was declared in the program. You can ignore this value and use whatever value is necessary for your purpose. **Note** You must return the size in bytes of the space you allocate. The library routine that calls _FTN _ALLOC() ensures that all other occurrences of this common block fit in the space you allocated. Return the size in bytes of the space you allocate by modifying the size parameter.
name	is the name of the common block being dynamically allocated.

Rules of Using Dynamic Common Option

The following are some limitations that you should be aware of when using the dynamic common option:

- If you use the technique of implementing your own allocation routine, then you should specify only one dynamic COMMON block on the command line. Otherwise, you may not know the name of the COMMON block for which you are allocating storage.
- An entity in a dynamic COMMON may not be initialized in a DATA statement.
- Only named COMMON blocks may be designated as dynamic COMMON.

- An entity in a dynamic `COMMON` must not be used in an `EQUIVALENCE` expression with an entity in a static `COMMON` or a `DATA`-initialized variable.

Compiler Optimizations

The variety of optimizations used by the Intel® Fortran Compiler enable you to enhance the performance of your application. Each optimization is performed by a set of options, see Compiler Options by Functional Groups Overview and Application Performance Optimizations Options section.

In addition to optimizations invoked by the compiler command line options, the compiler includes features which enhance your application performance such as directives, intrinsics, runtime library routines and various utilities. These features are discussed in the Optimization Support Features section.

Optimization Levels

Each of the command-line options: `-O`,`-O1`, `-O2` and `-O3` turn on several compiler capabilities. See the summary of these options.

The following table provides a summary of the optimizations that the compiler applies when you invoke
`-O`, `-O1` and/or `-O2`, or `-O3` optimizations.

Option	Optimization	Affected Aspect of Program
`-O1, -O2`	global register allocation	register use
`-O1, -O2`	instruction scheduling	instruction reordering
`-O1, -O2`	register variable detection	register use
`-O1, -O2`	common subexpression elimination	constants and expression evaluation
`-O1, -O2`	dead-code elimination	instruction sequencing
`-O1, -O2`	variable renaming	register use
`-O1, -O2`	copy propagation	register use
`-O1, -O2`	constant propagation	constants and expression evaluation
`-O1, -O2`	strength reduction-induction variable	simplification instruction, selection-sequencing
`-O1, -O2`	tail recursion elimination	calls, further optimization
`-O, -O2`	software pipelining for Itanium-based application	calls, further optimization
`-O2`	loop unrolling; inlining of intrinsics	calls, further optimization
`-O3`	prefetching, scalar replacement, loop transformations	memory access, instruction parallelism, predication, software pipelining

Setting Optimization Levels

For IA-32 and Itanium® architectures, these options behave in a different way. To specify the optimizations for your program, use options depending on the target architecture as explained in the tables that follow.

Itanium® Compiler

Option	Effect
-01	Optimizes to favor code size. Enables the same optimizations as -0 except for loop unrolling and software pipelining. At -01 the global code scheduler is tuned to favor code size.
-0, -02	Turn the software pipelining ON. Generally, -0 or -02 are recommended over -01.

IA-32 Compiler

Option	Effect
-0,-01,-02	Optimize to favor code speed. Disable option -fp. The -02 option is ON by default. Inlines intrinsics. Example: large database applications, code with many branches and not dominated by loops
-03	Enables -02 option with more aggressive optimization. Optimizes for maximum speed, but does not guarantee higher performance unless loop and memory access transformation take place. In conjunction with -axK and -xK options, this option causes the compiler to perform more aggressive data dependency analysis than for -02. This may result in longer compilation times.

IA-32 and Itanium Compilers

For IA-32 and Itanium architectures, the options can behave in a different way. To specify the optimizations for your program, use options depending on the target architecture as follows.

Option	Effect
-02	ON by default. -02 turns ON intrinsics inlining. Used for best overall performance on typical integer applications that do not make heavy use of floating point math. Enables the following capabilities for performance gain: • constant propagation

	• copy propagation • dead-code elimination • global register allocation • global instruction scheduling and control speculation • loop unrolling • optimized code selection • partial redundancy elimination • strength reduction/induction variable simplification • variable renaming • predication • software pipelining
`-O3`	Enables `-O2` option with more aggressive optimization. Optimizes for maximum speed, but may not improve performance for some programs. Used mostly for applications that make heavy use of floating-point calculations on large data sets.

Restricting Optimizations

The following options restrict or preclude the compiler's ability to optimize your program:

`-O0`	Disables optimizations `-O1`, `-O2`, and-or `-O3`. Enables `-fp` option.
`-mp`	Restricts optimizations that cause some minor loss or gain of precision in floating-point arithmetic to maintain a declared level of precision and to ensure that floating-point arithmetic more nearly conforms to the ANSI and IEEE* standards. See `-mp` option for more details.
`-nolib_inline`	Disables inline expansion of intrinsic functions.

For more information on ways to restrict optimization, see Interprocedural Optimizations with -Qoption.

Floating-point Arithmetic Precision

The options described in this section all provide optimizations with varying degrees of precision in floating-point (FP) arithmetic for IA-32 and Itanium® compiler. See the FP arithmetic precision options summary.

The `-mp and -mp1` options are used by both architectures. These options improve floating-point precision, but also affect the application performance. See more details about these options in Improving/Restricting FP Arithmetic Precision.

The FP options provide optimizations with varying degrees of precision in floating-point arithmetic. The option that disables these optimizations is `-O0`.

`-mp` Option

Use `-mp` to limit floating-point optimizations and maintain declared precision. For example, the Intel® Fortran Compiler can change floating-point division computations into multiplication by the reciprocal of the denominator. This change can alter the results of floating point division computations slightly. The `-mp` switch may slightly reduce execution speed. See Improving/Restricting FP Arithmetic Precision for more detail.

`-mp1` Option

Use the `-mp1` option to restrict floating-point precision to be closer to declared precision with less impact to performance than with the `-mp` option. The option will ensure the out-of-range check of operands of transcendental functions and improve accuracy of floating-point compares.

Floating-point Arithmetic Precision for IA-32 Systems

`-prec_div` Option

The Intel® Fortran Compiler can change floating-point division computations into multiplication by the reciprocal of the denominator. Use `-prec_div` to disable floating point division-to-multiplication optimization resulting in more accurate division results. May have speed impact.

`-pc{32|64|80}` Option

Use the `-pc{32|64|80}` option to enable floating-point significand precision control. Some floating-point algorithms, created for specific 32- and Itanium®-based systems, are sensitive to the accuracy of the significand or fractional part of the floating-point value. Use appropriate version of the option to round the significand to the number of bits as follows:

-pc32: 24 bits (single precision)

-pc64: 53 bits (double precision)

-pc80: 64 bits (extended precision)

The default version is -pc64 for full floating-point precision.

This option enables full optimization. Using this option does not have the negative performance impact of using the -mp option because only the fractional part of the floating-point value is affected. The range of the exponent is not affected.

 **Note**

This option only has effect when the module being compiled contains the main program.

 **Caution**

A change of the default precision control or rounding mode (for example, by using the -pc32 option or by user intervention) may affect the results returned by some of the mathematical functions.

Rounding Control, -rcd, -fp_port

The Intel Fortran Compiler uses the -rcd option to disable changing of rounding mode for floating-point-to-integer conversions.

The system default floating-point rounding mode is round-to-nearest. This means that values are rounded during floating-point calculations. However, the Fortran language requires floating-point values to be truncated when a conversion to an integer is involved. To do this, the compiler must change the rounding mode to truncation before each floating-point conversion and change it back afterwards.

The -rcd option disables the change to truncation of the rounding mode for all floating-point calculations, including floating-point-to-integer conversions. Turning on this option can improve performance, but floating-point conversions to integer will not conform to Fortran semantics.

You can also use the -fp_port option to round floating-point results at assignments and casts. This option has some speed impact.

Floating-point Arithmetic Precision for Itanium®-based Systems

The following Intel® Fortran Compiler options enable you to control the compiler optimizations for floating-point computations on Itanium®-based systems.

Contraction of FP Multiply and Add/Subtract Operations

`-IPF_fma[-]` enables or disables the contraction of floating-point multiply and add/subtract operations into a single operations. Unless `-mp` is specified, the compiler tries to contract these operations whenever possible. The `-mp` option disables the contractions.

`-IPF_fma` and `-IPF_fma-` can be used to override the default compiler behavior. For example, a combination of `-mp` and `-IPF_fma` enables the compiler to contract operations:

prompt>**efc -mp -IPF_fma *myprog.f***

FP Speculation

`-IPF_fp_speculation`*mode* sets the compiler to speculate on floating-point operations in one of the following *mode*s:

fast: sets the compiler to speculate on floating-point operations; this is the default.

safe: enables the compiler to speculate on floating-point operations only when it is safe;

strict: enables the compiler's speculation on floating-point operations preserving floating-point status in all situations. In the current version, this mode disables the speculation of floating-point operations (same as `off`).

off: disables the speculation on floating-point operations.

FP Operations Evaluation

`-IPF_flt_eval_method{0|2}` option directs the compiler to evaluate the expressions involving floating-point operands in the following way:

`-IPF_flt_eval_method0` directs the compiler to evaluate the expressions involving floating-point operands in the precision indicated by the variable types declared in the program.

`-IPF_flt_eval_method2` is not supported in the current version.

Controlling Accuracy of the FP Results

`-IPF_fltacc[-]` enables the compiler to apply optimizations that affect floating-point accuracy. The default is `-IPF_fltacc-`.

The Itanium® compiler may reassociate floating-point expressions to improve application performance. Use `-IPF_fltacc` or `-mp` to disable this behavior.

Improving/Restricting FP Arithmetic Precision

The `-mp` and `-mp1` options maintain and restrict, respectively, floating-point precision, but also affect the application performance. The `-mp1` option causes less impact on performance than the `-mp` option. `-mp1` ensures the out-of-range check of operands of transcendental functions and improve accuracy of floating-point compares.

The `-mp` option restricts some optimizations to maintain declared precision and to ensure that floating-point arithmetic conforms more closely to the ANSI and IEEE* standards. This option causes more frequent stores to memory, or disallow some data from being register candidates altogether. The Intel architecture normally maintains floating point results in registers. These registers are 80 bits long, and maintain greater precision than a double-precision number. When the results have to be stored to memory, rounding occurs. This can affect accuracy toward getting more of the "expected" result, but at a cost in speed. The `-pc{32|64|80}` option (IA-32 only) can be used to control floating point accuracy and rounding, along with setting various processor IEEE flags.

For most programs, specifying this option adversely affects performance. If you are not sure whether your application needs this option, try compiling and running your program both with and without it to evaluate the effects on performance versus precision.

Specifying this option has the following effects on program compilation:

- On **IA-32 systems**, floating-point user variables declared as floating-point types are not assigned to registers.
- On **Itanium®-based systems**, floating-point user variables may be assigned to registers. The expressions are evaluated using precision of source operands. The compiler will not use Floating-point Multiply and Add (FMA) function to contract multiply and add/subtract operations in a single operation. The contractions can be enabled by using `-IPF_fma` option. The compiler will not speculate on floating-point operations that may affect the floating-point state of the machine. See Floating-point Arithmetic Precision for Itanium-based Systems.
- Floating-point arithmetic comparisons conform to IEEE 754.
- The exact operations specified in the code are performed. For example, division is never changed to multiplication by the reciprocal.
- The compiler performs floating-point operations in the order specified without reassociation.
- The compiler does not perform the constant folding on floating-point values. Constant folding also eliminates any multiplication by 1, division by 1, and addition or subtraction of 0. For example, code that adds 0.0 to a number is executed exactly as written. Compile-time floating-point arithmetic is not performed to ensure that floating-point exceptions are also maintained.

For IA-32 systems, whenever an expression is spilled, it is spilled as 80 bits (EXTENDED PRECISION), not 64 bits (DOUBLE PRECISION). Floating-point operations conform to IEEE 754. When assignments to type REAL and DOUBLE PRECISION are made, the precision is rounded from 80 bits (EXTENDED) down to 32 bits (REAL) or 64 bits (DOUBLE PRECISION). When you do not specify -O0, the extra bits of precision are not always rounded away before the variable is reused.

- Even if vectorization is enabled by the -xK option, the compiler does not vectorize reduction loops (loops computing the dot product) and loops with mixed precision types. Similarly, the compiler does not enable certain loop transformations. For example, the compiler does not transform reduction loops to perform partial summation or loop interchange.

Targeting a Processor and Extensions Support

This section describes targeting a processor and processor dispatch options, the feature for IA-32 only. The options -tpp{5|6|7} optimizes for the IA-32 processors, and the options -tpp1 and -tpp2 optimize for the Itanium® processor family. The options -x{i|M|K|W} and -ax{i|M|K|W} provide support to generate code that is specific to processor-instruction extensions. See the summary of options supporting Targeting a Processor and Extensions Support.

-tpp{1|2}	-tpp1—Itanium® processor -tpp2—Itanium® 2 processor
-tpp{5|6|7}	-tpp5—Pentium® processor. -tpp6—Pentium® Pro, Pentium® II, and Pentium® III processors. -tpp7—Pentium® 4 and Xeon(TM) processors. Requires the RedHat* version 7.1 and support of Streaming SIMD Extensions 2. Default
-x{i|M|K|W}	Generates specialized code to run exclusively on the processors supporting the extensions indicated by the i, M, K, W codes.
-ax{i|M|K|W}	Generates specialized code to run exclusively on the processors supporting the extensions indicated by the i, M, K, W codes while also generating generic IA-32 code.

For example, on Pentium® III processor, if you have mostly integer code and only a small portion of floating-point code, you may want to compile with -axM rather than -axK because MMX(TM) technology extensions perform the best with the integer data.

The -ax and -x options are backward compatible with the extensions supported. On Intel®

Pentium® 4 and Xeon processors, you can gear your code to any of the previous processors specified by `K`, `M`, or `i`.

Targeting a Processor, `-tpp{n}`

The Intel® Fortran Compiler lets you choose whether to optimize the performance of your application for specific processors or to ensure your application can execute on a range of processors.

Optimizing for a Specific Processor Without Excluding Others

Use the `-tpp{n}` option to optimize your application's performance for specific processors. Regardless of which `-tpp{n}` suboption you choose, your application is optimized to use all the benefits of that processor with the resulting binary file still capable of running on any of the processors listed.

To optimize for...	Use...
Itanium® processor	`-tpp2` (Itanium-based systems)
Itanium® 2 processor.	`-tpp2` (default for Itanium-based systems)
Pentium® processor and Pentium® processor with MMX(TM) technology	`-tpp5`
Pentium® Pro, Pentium® II and Pentium® III processors	`-tpp6`
Intel® Pentium® 4 and Xeon(TM) processors	`-tpp7` (default for IA-32 systems)

For example, the following commands compile and optimize the source program `prog.f` for the Pentium® 4 processor:

prompt>**ifc *prog.f***

prompt>**ifc -tpp7 *prog.f***

By default, the Itanium® compiler targets optimization to the Itanium 2 processor as recommended for the best performance on Itanium® processor systems. The generated code is compatible with the Intel® Itanium® 2 processor.

prompt>**efc *prog.f***

The above command targets optimization to the Itanium 2 processor. However if you intend to target your application specifically to the Intel® Itanium® processor, use the `-tpp1` option:

prompt>**efc -tpp1 *prog.f***

Exclusive Specialized Code with -x{i|M|K|W}

The -x{i|M|K|W} option specifies the minimum set of processor extensions required to exist on processors on which you execute your program as follows:

i Pentium® Pro, Pentium II processors
M Pentium® with MMX(TM) technology processor
K Pentium® III processor
W Pentium® 4 and Xeon(TM) processors.

The resulting code can contain unconditional use of the specified processor extensions. When you use
-x{i|M|K|W}, the code generated by the compiler might not execute correctly on IA-32 processors that lack the specified extensions.

The following example compiles the program myprog.f, using the i extension. This means the program will require Pentium Pro, Pentium II processors, and later architectures to execute.

prompt>**ifc -O2 -tpp6 -xi *myprog.f***

The resulting program, myprog, might not execute on a Pentium processor, but will execute on Pentium® Pro, Pentium II, and Pentium III processors.

> Caution
> If a program compiled with -x{i|M|K|W} is executed on a processor that lacks the specified extensions, it can fail with an illegal instruction exception, or display other unexpected behavior.

-x Summary

To Optimize for...	Use this option
Pentium Pro and Pentium II processors, which use the CMOV and FCMOV, and FCOMI instructions	-xi
Pentium processors with MMX(TM) technology instructions	-xM
Pentium III processor with the Streaming SIMD Extensions, implies i and M instructions	-xK
Pentium 4 and Xeon processors with the Streaming SIMD Extensions 2, implies i, M, and K instructions	-xW

You can specify more than one code with the -x option. For example, if you specify -xMK, the compiler will decide whether the resulting executable will benefit better from the MMX technology (M) or the Streaming SIMD Extensions (K). It is the developer's responsibility to

use the option's version corresponding to the processor generation.

Specialized Code with -ax{i|M|K|W}

With -ax{i|M|K|W} you can instruct the compiler to compile your application so that processor-specific extensions are included in the compilation but only used if the processor supports them as follows:

i Pentium® Pro, Pentium® II processors
M Pentium® with MMX™ technology processor
K Pentium® III processor
W Pentium® 4 and Xeon processors

When the compiled application is run, it detects the extensions supported by the processor.

- If the processor supports the specialized extensions, the extensions are executed.
- If the processor does not support the specialized code, the extensions are not executed and a more generic version of the code is executed instead.

Applications compiled with -ax{i|M|K|W} have increased code size, but the performance of such code is better than standard optimized code, although slightly slower than if compiled with the -x{i|M|K|W} due to the latter's smaller overhead of checking for which processor the application is being run on.

Note

Applications that you compile to optimize themselves for specific processors in this way will execute on any Intel 32-bit processor. Such compilations are, however subject to any exclusive specialized code restrictions you impose during compilation with the -x option.

-ax Summary

To Optimize for...	Use this option
Pentium® Pro and Pentium II processors, which use the CMOV and FCMOV, and FCOMI instructions	-axi
Pentium processors with MMX(TM) technology instructions	-axM
Pentium III processor with the Streaming SIMD Extensions, implies i and M instructions	-axK
Pentium 4 processor with the Streaming SIMD Extensions 2, implies i, M, and K instructions	-axW

Checking for Performance Gain

The `-ax{i|M|K|W}` option directs the compiler to find opportunities to generate special versions of functions that use instructions supported on the specified processors. If the compiler finds such an opportunity, it first estimates whether generating a processor-specific version of a function results in a performance gain. If this is the case, the compiler generates both a processor-specific version of a function and a generic version of this function that will run on any IA-32 architecture processor.

You can specify more than one code with the `-ax` option. For example, if you specify `-axMK`, the compiler will decide whether the resulting executable will benefit better from the MMX technology (`M`) or the Streaming SIMD Extensions (`K`). At runtime, one of the two versions is chosen to execute depending on the processor the program is currently running on. In this way, the program can get large performance gains on more advanced processors, while still working properly on older processors. It is the developer's responsibility to use the option's version corresponding to the processor generation.

The disadvantages of using `-ax{i|M|K|W}` are:

- The size of the binary increases because it contains processor-specific and generic versions of the code.
- The runtime checks to determine which code to run slightly affect performance.

Combining Processor Target and Dispatch Options

The following table shows how to combine processor target and dispatch options to compile applications with different optimizations and exclusions.

Optimize exclusively for...	**...while optimizing without exclusion for...**					
	Pentium® Processor	**Pentium® Processor with MMX (TM) technology**	**Pentium® Pro Processor**	**Pentium® II Processor**	**Pentium® III Processor**	**Penti 4, Xe (TM) Proce**
Pentium Processor	`-tpp5`	`-tpp5`	`-tpp6`	`-tpp6`	`-tpp6`	`-tpp`
Pentium Processor with MMX technology	N-A	`-tpp5, -xM`	`-tpp6`	`-tpp6, -xM`	`-tpp6, -xM`	`-tpp -xM`
Pentium Pro Processor	N-A	N-A	`-tpp6, -xi`	`-tpp6, -xi`	`-tpp6, -xi`	`-tpp -xi`
Pentium II Processor	N-A	N-A	N-A	`-tpp6, -xiM`	`-tpp6, -xiM`	`-tpp -xiM`

Pentium III Processor	N-A	N-A	N-A	N-A	`-tpp6, -xK`	`-tpp -xK`
Pentium 4, Xeon Processors	N-A	N-A	N-A	N-A	N-A	`-tpp -xW`

Example of -x and -ax Combinations

If you wanted your application to

- always require the MMX technology extensions
- use Pentium Pro processor extensions when the processor it is run on offers it, and to not use them when it does not

you could generate such an application with the following command line:

prompt>**ifc -02 -tpp6 -xM -xi *myprog.f***

`-xM` above restricts the application to running on Pentium processors with MMX technology or later processors. If you wanted to enable the application to run on earlier generations of Intel® IA-32 processors as well, you would use the following command line:

prompt>**ifc -02 -tpp6 -axM *myprog.f***

Interptocedural Optimizations

Use `-ip` and `-ipo` to enable interprocedural optimizations (IPO), which enable the compiler to analyze your code to determine where you can benefit from the optimizations listed in tables that follow. See IPO options summary.

IA-32 and Itanium®-based applications

Optimization	Affected Aspect of Program
inline function expansion	calls, jumps, branches, and loops
interprocedural constant propagation	arguments, global variables, and return values
monitoring module-level static variables	further optimizations, loop invariant code
dead code elimination	code size
propagation of function characteristics	call deletion and call movement
multifile optimization	affects the same aspects as `-ip`, but across multiple files

IA-32 applications only

Optimization	Affected Aspect of Program
passing arguments in registers	calls, register usage
loop-invariant code motion	further optimizations, loop invariant code

Inline function expansion is one of the main optimizations performed by the interprocedural optimizer. For function calls that the compiler believes are frequently executed, the compiler might decide to replace the instructions of the call with code for the function itself.

With `-ip`, the compiler performs inline function expansion for calls to procedures defined within the current source file. However, when you use `-ipo` to specify multifile IPO, the compiler performs inline function expansion for calls to procedures defined in separate files.

To disable the IPO optimizations, use the `-O0` option.

Multifile IPO

Multifile IPO obtains potential optimization information from individual program modules of a multifile program. Using the information, the compiler performs optimizations across modules.

Building a program is divided into two phases: compilation and linkage. Multifile IPO performs different work depending on whether the compilation, linkage or both are performed.

Compilation Phase

As each source file is compiled, multifile IPO stores an intermediate representation (`IR`) of the source code in the object file, which includes summary information used for optimization.

By default, the compiler produces "mock" object files during the compilation phase of multifile IPO. Generating mock files instead of real object files reduces the time spent in the multifile IPO compilation phase. Each mock object file contains the `IR` for its corresponding source file, but no real code or data. These mock objects must be linked using the `-ipo` option in `ifc`/`efc` or using the `xild` tool. (See Creating a Multifile IPO Executable with xild.)

Note

Failure to link "mock" objects with `ifc`/`efc` and `-ipo` or `xild` will result in linkage errors. There are situations where mock object files cannot be used. See Compilation with Real Object Files for more information.

Linkage Phase

When you specify `-ipo`, the compiler is invoked a final time before the linker. The compiler performs multifile IPO across all object files that have an IR.

> **Note**
> The compiler does not support multifile IPO for static libraries (`.a` files). See Compilation with Real Object Files for more information.

`-ipo` enables the driver and compiler to attempt detecting a whole program automatically. If a whole program is detected, the interprocedural constant propagation, stack frame alignment, data layout and padding of common blocks perform more efficiently, while more dead functions get deleted. This option is safe.

Creating a Multifile IPO Executable with Command Line

Enable multifile IPO for compilations targeted for IA-32 architecture and for compilations targeted for Itanium® architecture as follows in the example below.

Compile your source files with `-ipo` as follows:

Compile source files to produce object files:
prompt>**ifc -ipo -c *a.f b.f c.f***

Produces `a.o`, `b.o`, and `c.o` object files containing Intel compiler intermediate representation (`IR`) corresponding to the compiled source files `a.f`, `b.f`, and `c.f`. Using `-c` to stop compilation after generating `.o` files is required. You can now optimize interprocedurally.

Link object files to produce application executable:
prompt>**ifc -o*ipo_file* -ipo *a.o b.o c.o***

The `ifc` command performs IPO for objects containing `IR` and creates a new list of object (s) to be linked. The `ifc` command calls GCC `ld` to link the specified object files and produce `ipo_file.exe` specified by the `-o` option. Multifile IPO is applied only to the source files that have an `IR`, otherwise the object file passes to link stage.

The `-o`*name* option stores the executable in `ipo_file`. Multifile IPO is applied only to the source files that have an IR, otherwise the object file passes to link stage.

For efficiency, combine steps 1 and 2:

prompt>**ifc -ipo -o*ipo_file a.f b.f c.f***

For Itanium®-based applications, use the same steps with the `efc` command.

Instead of `ifc` or `efc`, you can use the `xild` tool.

For a description of how to use multifile IPO with profile information for further optimization,

see Example of Profile-Guided Optimization.

Creating a Multifile IPO Executable Using `xild`

Use the Intel® linker, `xild`, instead of step 2 in Creating a Multifile IPO Executable with Command Line. The Intel linker `xild` performs the following steps:

1. Invokes the Intel compiler to perform multifile IPO if objects containing `IR` are found.
2. Invokes GCC `ld` to link the application.

The command-line syntax for `xild` is the same as that of the GCC linker:

prompt>**xild [<options>] <LINK_commandline>**

where:

- `[<options>]` (optional) may include any GCC linker options or options supported only by `xild`.
- `<LINK_commandline>` is your linker command line containing a set of valid arguments to the `ld`.

To place the multifile IPO executable in `ipo_file`, use the option `-o`*`filename`*, for example:

prompt>**xild -o*ipo_file a.o b.o c.o***

`xild` calls Intel compiler to perform IPO for objects containing `IR` and creates a new list of object(s) to be linked. Then `xild` calls `ld` to link the object files that are specified in the new list and produce `ipo_file` executable specified by the `-o`*`filename`* option.

Note

The `-ipo` option can reorder object files and linker arguments on the command line. Therefore, if your program relies on a precise order of arguments on the command line, `-ipo` can affect the behavior of your program.

Usage Rules

You must use the Intel linker `xild` to link your application if:

- Your source files were compiled with multifile IPO enabled. Multifile IPO is enabled by specifying the `-ipo` command-line option
- You normally would invoke the GCC linker (`ld`) to link your application.

The `xild` Options

The additional options supported by `xild` may be used to examine the results of multifile IPO. These options are described in the following table.

`-qipo_fa[`*`file.s`*`]`	Produces assembly listing for the multifile IPO compilation. You may specify an optional name for the listing file, or a directory (with the backslash) in which to place the file. The default listing name is `ipo_out.s`.
`-qipo_fo[`*`file.o`*`]`	Produces object file for the multifile IPO compilation. You may specify an optional name for the object file, or a directory (with the backslash) in which to place the file. The default object file name is `ipo_out.o`.
`-ipo_fcode-asm`	Add code bytes to assembly listing
`-ipo_fsource-asm`	Add high-level source code to assembly listing
`-ipo_fsource-asm, -ipo_fnoverbose-asm`	Enable and disable, respectively, inserting comments containing version and options used in the assembly listing for `xild`.

Compilation with Real Object Files

In certain situations you might need to generate real object files with `-ipo`. To force the compiler to produce real object files instead of "mock" ones with IPO, you must specify `-ipo_obj` in addition to `-ipo`.

Use of `-ipo_obj` is necessary under the following conditions:

- The objects produced by the compilation phase of `-ipo` will be placed in a static library without the use of `xiar`. The compiler does not support multifile IPO for static libraries, so all static libraries are passed to the linker. Linking with a static library that contains "mock" object files will result in linkage errors because the objects do not contain real code or data. Specifying `-ipo_obj` causes the compiler to generate object files that can be used in static libraries.

- Alternatively, if you create the static library using `xiar`, then the resulting static library will work as a normal library.

- The objects produced by the compilation phase of `-ipo` might be linked without the `-ipo` option and without the use of `xiar`.

- You want to generate an assembly listing for each source file (using `-S`) while compiling with `-ipo`. If you use `-ipo` with `-S`, but without `-ipo_obj`, the compiler

issues a warning and an empty assembly file is produced for each compiled source file.

Creating a Library from IPO Objects

Normally, libraries are created using a library manager such as `ar`. Given a list of objects, the library manager will insert the objects into a named library to be used in subsequent link steps.

```
prompt>xiar cru user.a a.obj b.obj
```

The above command creates a library named `user.a` that contains the `a.o` and `b.o` objects.

If, however, the objects have been created using `-ipo -c`, then the objects will not contain a valid object but only the intermediate representation (`IR`) for that object file. For example:

```
prompt>ifc -ipo -c a.f b.f
```

will produce `a.o` and `b.o` that only contains `IR` to be used in a link time compilation. The library manager will not allow these to be inserted in a library.

In this case you must use the Intel library driver `xild -ar`. This program will invoke the compiler on the `IR` saved in the object file and generate a valid object that can be inserted in a library.

```
prompt>xild -lib cru user.a a.o b.o
```

See Creating a Multifile IPO Executable Using `xild`.

Analyzing the Effects of Multifile IPO, `-ipo_c`, `-ipo_S`

The `-ipo_c` and `-ipo_S` options are useful for analyzing the effects of multifile IPO, or when experimenting with multifile IPO between modules that do not make up a complete program.

Use the `-ipo_c` option to optimize across files and produce an object file. This option performs optimizations as described for `-ipo`, but stops prior to the final link stage, leaving an optimized object file. The default name for this file is `ipo_out.o`. You can use the `-o` option to specify a different name. For example:

```
prompt>ifc -tpp6 -ipo_c -ofilename a.f b.f c.f
```

Use the `-ipo_S` option to optimize across files and produce an assembly file. This option performs optimizations as described for `-ipo`, but stops prior to the final link stage, leaving an optimized assembly file. The default name for this file is `ipo_out.s`. You can use the `-o` option to specify a different name. For example:

```
prompt>ifc -tpp6 -ipo_S -ofilename a.f b.f c.f
```

For more information on inlining and the minimum inlining criteria, see Criteria for Inline Function Expansion and Controlling Inline Expansion of User Functions.

Using `-ip` with `-Qoption` Specifiers

You can adjust the Intel® Fortran Compiler's optimization for a particular application by experimenting with memory and interprocedural optimizations.

Enter the `-Qoption` option with the applicable keywords to select particular inline expansions and loop optimizations. The option must be entered with a `-ip` or `-ipo` specification, as follows:

`-ip[-Qoption,tool,opts]`
where `tool` is Fortran (`f`) and `opts` are `-Qoption` specifiers (see below). Also refer to Criteria for Inline Function Expansion to see how these specifiers may affect the inlining heuristics of the compiler.

See Passing Options to Other Tools (-Qoption,tool,opts) for details about `-Qoption`.

`-Qoption` Specifiers

If you specify -ip or -ipo without any `-Qoption` qualification, the compiler

- expands functions in line
- propagates constant arguments
- passes arguments in registers
- monitors module-level static variables.

You can refine interprocedural optimizations by using the following `-Qoption` specifiers. To have an effect, the `-Qoption` option must be entered with either `-ip` or `-ipo` also specified, as in this example:

`-ip -Qoption,f,ip_specifier`

where `ip_specifier is one of the -Qoption specifiers described in the table that follows.`

-Qoption Specifiers	
`-ip_args_in_regs=0`	Disables the passing of arguments in registers. By default, external functions can pass arguments in registers when called locally. Normally, only static functions can pass arguments in registers, provided the address of the function is not taken and the function does not use a variable number of arguments.
`-ip_ninl_max_stats=n`	Sets the valid number of intermediate language statements for a function that is expanded in line. The number n is a positive integer. The number of intermediate language statements usually exceeds the actual number of source language statements. The default value for n is 230.
`-ip_ninl_min_stats=n`	Sets the valid min number of intermediate language statements for a function that is expanded in line. The number n is a positive integer. The default value for `ip_ninl_min_stats` is: **IA-32 compiler**: `ip_ninl_min_stats` = 7 **Itanium® compiler**: `ip_ninl_min_stats` = 15
`-ip_ninl_max_total_stats=n`	Sets the maximum increase in size of a function, measured in intermediate language statements, due to inlining. The number n is a positive integer. The default value for n is 2000.

The following command activates procedural and interprocedural optimizations on source.f and sets the maximum increase in the number of intermediate language statements to five for each function:

`prompt>`**`ifc -ip -Qoption`*`f,-ip_ninl_max_stats`*`=5 `*`source.f`**

Criteria for Inline Function Expansion

For a routine to be considered for inlining, it has to meet certain minimum criteria. There are criteria to be met by the call-site, the caller, and the callee. The call-site is the site of the call to the function that might be inlined. The caller is the function that contains the call-site. The callee is the function being called that might be inlined.

Minimum call-site criteria:

- The number of actual arguments must match the number of formal arguments of the callee.
- The number of return values must match the number of return values of the callee.
- The data types of the actual and formal arguments must be compatible.
- No multilingual inlining is permitted. Caller and callee must be written in the same source language.

Minimum criteria for the caller:

- At most 2000 intermediate statements will be inlined into the caller from all the call-sites being inlined into the caller. You can change this value by specifying the option

  `-Qoption`*`f,-ip_inline_max_total_stats`*`=new value`

- The function must be called if it is declared as static. Otherwise, it will be deleted.

Minimum criteria for the callee:

- Does not have variable argument list.
- Is not considered infrequent due to the name. Routines which contain the following substrings in their names are not inlined: `abort`, `alloca`, `denied`, `err`, `exit`, `fail`, `fatal`, `fault`, `halt`, `init`, `interrupt`, `invalid`, `quit`, `rare`, `stop`, `timeout`, `trace`, `trap`, and `warn`.
- Is not considered unsafe for other reasons.

Selecting Routines for Inlining

Once these criteria are met, the compiler picks the routines whose inline expansions will provide the greatest benefit to program performance. This is done using the default heuristics. The inlining heuristics used by the compiler differ based on whether you use profile-guided optimizations (`-prof_use`) or not.

When you use profile-guided optimizations with `-ip` or `-ipo`, the compiler uses the following heuristics:

- The default heuristic focuses on the most frequently executed call sites, based on the profile information gathered for the program.
- By default, the compiler does not inline functions with more than 230 intermediate statements. You can change this value by specifying the option `-Qoption,`*`f,/ip_ninl_max_stats`*`=new value`.

- The default inline heuristic will stop inlining when direct recursion is detected.
- The default heuristic always inlines very small functions that meet the minimum inline criteria.

Default for Itanium®-based applications: `ip_ninl_min_stats = 15`.

Default for IA-32 applications: `ip_ninl_min_stats = 7`.

These limits can be modified with the option `-Qoption,f,/ip_ninl_min_stats=new value`. See
-Qoption Specifiers and Profile-Guided Optimization (PGO).

When you do not use profile-guided optimizations with `-ip` or `-ipo`, the compiler uses less aggressive inlining heuristics: it inlines a function if the inline expansion does not increase the size of the final program.

Controlling Inline Expansion of User Functions

The compiler enables you to control the amount of inline function expansion, with the options shown in the following summary.

Option	Effect
`-ip_no_inlining`	This option is only useful if `-ip` or `-ipo` is also specified. In such case, `-ip_no_inlining` disables inlining that would result from the `-ip` interprocedural optimizations, but has no effect on other interprocedural optimizations.
`-inline_debug_info`	Preserve the source position of inlined code instead of assigning the call-site source position to inlined code.
IA-32 only: `-ip_no_pinlining`	Disables partial inlining; can be used if `-ip` or `-ipo` is also specified.
`-Ob{0|1|2}`	Controls the compiler's inline expansion. The amount of inline expansion performed varies as follows: `-Ob0`: disables inline expansion of user-defined functions `-Ob1`: disables inlining unless `-ip` or `-Ob2` is specified. Enables inlining of functions.

	`-Ob2`: Enables inlining of any function. However, the compiler decides which functions are inlined. This option enables interprocedural optimizations and has the same effect as specifying the `-ip` option.

Inline Expansion of Library Functions

By default, the compiler automatically expands (inlines) a number of standard and math library functions at the point of the call to that function, which usually results in faster computation.

However, the inlined library functions do not set the `errno` variable when being expanded inline. In code that relies upon the setting of the `errno` variable, you should use the `-nolib_inline` option. Also, if one of your functions has the same name as one of the compiler-supplied library functions, then when this function is called, the compiler assumes that the call is to the library function and replaces the call with an inlined version of the library function.

So, if the program defines a function with the same name as one of the known library routines, you must use the `-nolib_inline` option to ensure that the user-supplied function is used.
`-nolib_inline` disables inlining of all intrinsics.

Your results can vary slightly using the preceding optimizations.

> **Note**
> Automatic inline expansion of library functions is not related to the inline expansion that the compiler does during interprocedural optimizations. For example, the following command compiles the program `sum.f` without expanding the math library functions:
>
> **IA-32 applications:**
>
> *prompt*>**ifc -ip -nolib_inline *sum.f***
>
> **Itanium®-based applications:**
>
> *prompt*>**efc -ip -nolib_inline *sum.f***

For information on the Intel-provided intrinsic functions, see Additional Intrinsic Functions in the Reference section.

Profile-guided Optimizations

Profile-guided optimizations (PGO) tell the compiler which areas of an application are most frequently executed. By knowing these areas, the compiler is able to be more selective and specific in optimizing the application. For example, the use of PGO often enables the compiler to make better decisions about function inlining, thereby increasing the effectiveness of interprocedural optimizations. See PGO Options summary.

Instrumented Program

Profile-guided Optimization creates an instrumented program from your source code and special code from the compiler. Each time this instrumented code is executed, the instrumented program generates a dynamic information file. When you compile a second time, the dynamic information files are merged into a summary file. Using the profile information in this file, the compiler attempts to optimize the execution of the most heavily travelled paths in the program.

Unlike other optimizations such as those strictly for size or speed, the results of IPO and PGO vary. This is due to each program having a different profile and different opportunities for optimizations. The guidelines provided help you determine if you can benefit by using IPO and PGO. You need to understanding the principles of the optimizations and the unique aspects of your source code.

Added Performance with PGO

In this version of the Intel® Fortran Compiler, PGO is improved in the following ways:

- Register allocation uses the profile information to optimize the location of spill code.
- For indirect function calls, branch prediction is improved by identifying the most likely targets. With the Pentium® 4 and Xeon(TM) processors' longer pipeline, improving branch prediction translates into high performance gains.
- The compiler detects and does not vectorize loops that execute only a small number of iterations, reducing the run time overhead that vectorization might otherwise add.

Profile-guided Optimizations Methodology

PGO works best for code with many frequently executed branches that are difficult to predict at compile time. An example is the code with intensive error-checking in which the error conditions are false most of the time. The "cold" error-handling code can be placed such that the branch is hardly ever mispredicted. Minimizing "cold" code interleaved into the "hot" code improves instruction cache behavior.

PGO Phases

The PGO methodology requires three phases:

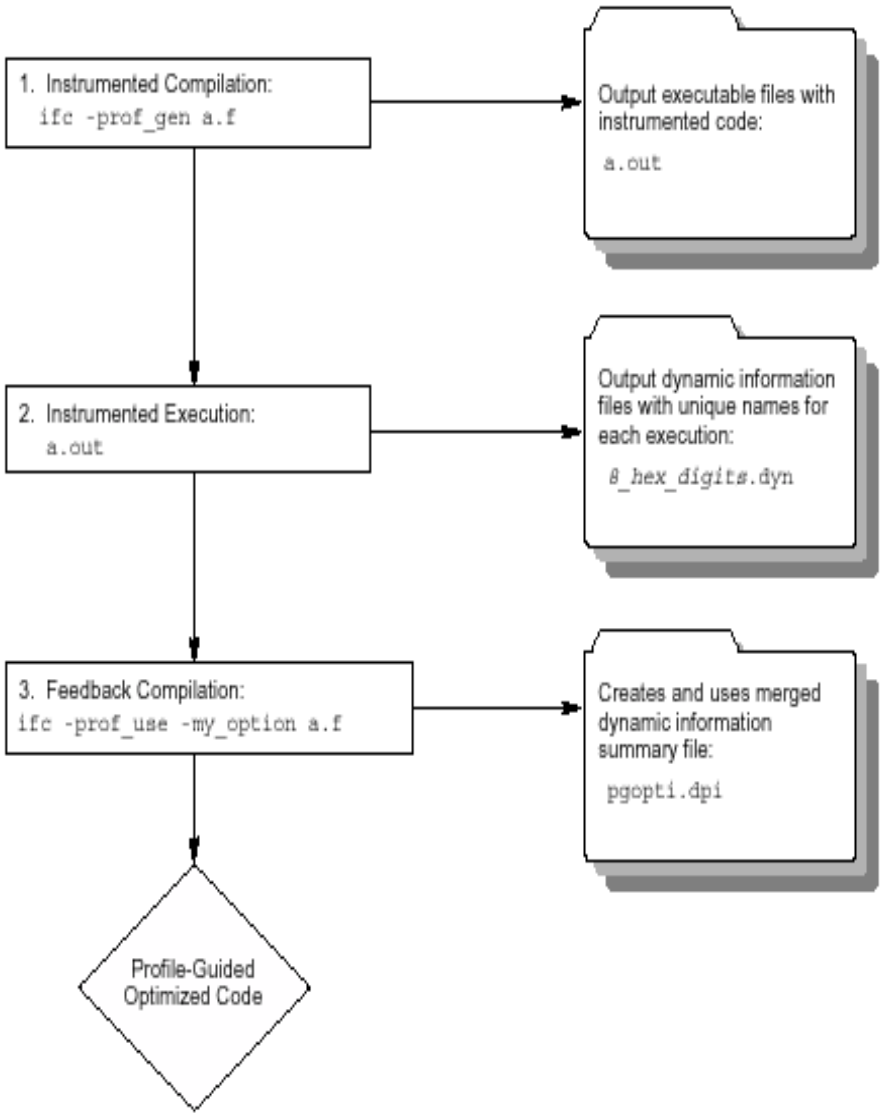
1. **Instrumentation compilation** and linking with `-prof_gen`

2. **Instrumented execution** by running the executable; as a result, the dynamic-information files (`.dyn`) are produced.

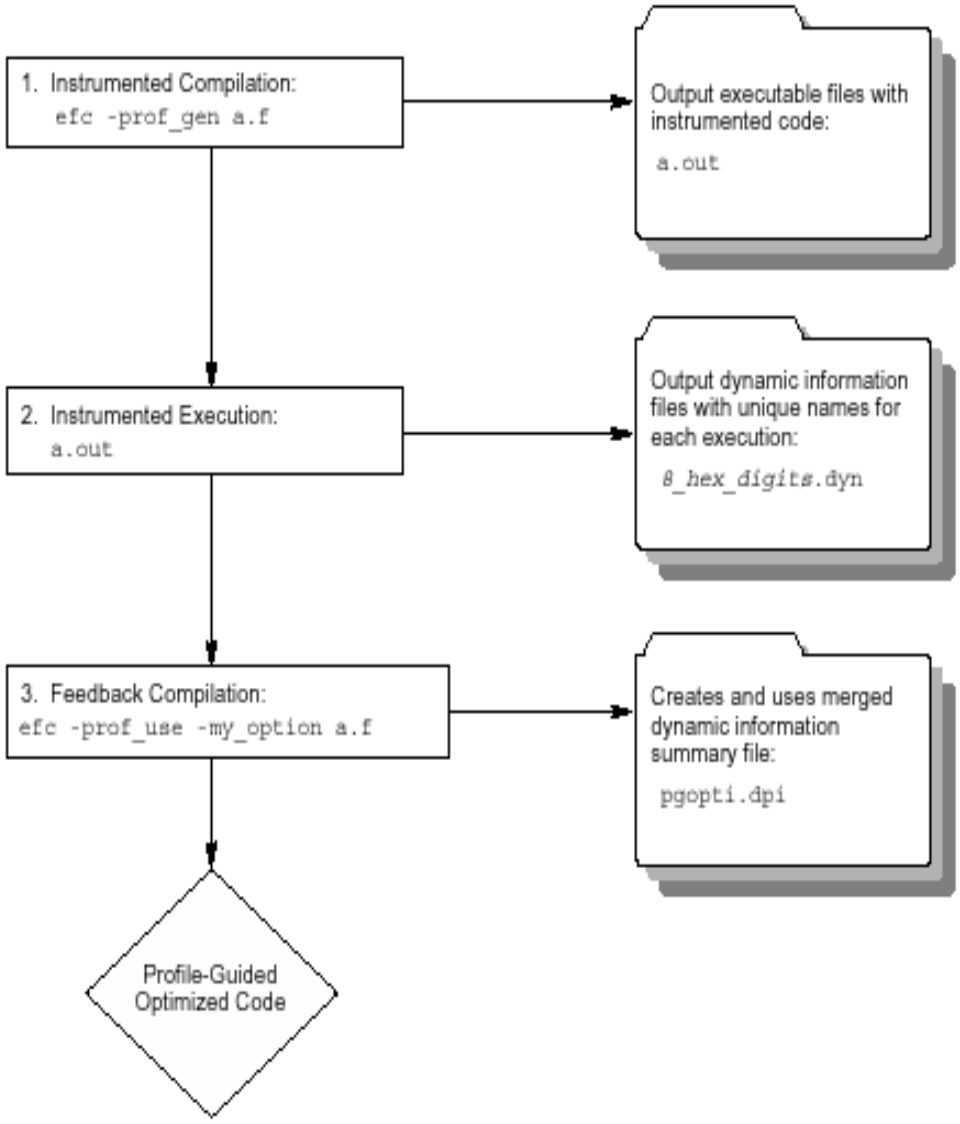
3. **Feedback compilation** with `-prof_use`

The flowcharts below illustrate this process for IA-32 compilation and Itanium®-based compilation. A key factor in deciding whether you want to use PGO lies in knowing which sections of your code are the most heavily used. If the data set provided to your program is very consistent and it elicits a similar behavior on every execution, then PGO can probably help optimize your program execution. However, different data sets can elicit different algorithms to be called. This can cause the behavior of your program to vary from one execution to the next.

IA-32 Phases of Basic Profile-Guided Optimization



Phases of Basic Profile-Guided Optimization for Itanium®-based applications



Basic PGO Options

The options used for basic PGO optimizations are:

- `-prof_gen[x]` for generating instrumented code
- `-prof_use` for generating a profile-optimized executable

In cases where your code behavior differs greatly between executions, you have to ensure that the benefit of the profile information is worth the effort required to maintain up-to-date profiles. In the basic profile-guided optimization, the following options are used in the phases of the PGO:

Generating Instrumented Code, `-prof_gen[x]`

The `-prof_gen[x]` option instruments the program for profiling to get the execution count of each basic block. It is used in phase 1 of the PGO to instruct the compiler to produce

of each basic block. It is used in phase 1 of the PGO to instruct the compiler to produce instrumented code in your object files in preparation for instrumented execution. Parallel `make` is automatically supported for `-prof_genx` compilations.

Generating a Profile-optimized Executable, `-prof_use`

The `-prof_use` option is used in phase 3 of the PGO to instruct the compiler to produce a profile-optimized executable and merges available dynamic-information (`.dyn`) files into a `pgopti.dpi` file.

> **Note:**
>
> The dynamic-information files are produced in phase 2 when you run the instrumented executable.

If you perform multiple executions of the instrumented program, `-prof_use` merges the dynamic-information files again and overwrites the previous `pgopti.dpi` file.

Disabling Function Splitting, `-fnsplit-` (Itanium® Compiler only)

`-fnsplit-` disables function splitting. Function splitting is enabled by `-prof_use` in phase 3 to improve code locality by splitting routines into different sections: one section to contain the cold or very infrequently executed code and one section to contain the rest of the code (hot code).

You can use `-fnsplit-` to disable function splitting for the following reasons:

- Most importantly, to get improved debugging capability. In the debug symbol table, it is difficult to represent a split routine, that is, a routine with some of its code in the hot code section and some of its code in the cold code section.

  The `-fnsplit-` option disables the splitting within a routine but enables function grouping, an optimization in which entire routines are placed either in the cold code section or the hot code section. Function grouping does not degrade debugging capability.

- Another reason can arise when the profile data does not represent the actual program behavior, that is, when the routine is actually used frequently rather than infrequently.

> **Note**
>
> For Itanium®-based applications, if you intend to use the `-prof_use` option with optimizations at the `-O3` level, the `-O3` option must be on. If you intend to use the `-prof_use` option with optimizations at the `-O2` level or lower, you can generate the profile data with the default options.

See an example of using PGO.

Advanced PGO Options

Advanced PGO Options

The options controlling advanced PGO optimizations are:

- `-prof_dir`*`dirname`*
- `-prof_file`*`filename`*.

Specifying the Directory for Dynamic Information Files

Use the `-prof_dir`*`dirname`* option to specify the directory in which you intend to place the dynamic information (`.dyn`) files to be created. The default is the directory where the program is compiled. The specified directory must already exist.

You should specify `-prof_dir`*`dirname`* option with the same directory name for both the instrumentation and feedback compilations. If you move the `.dyn` files, you need to specify the new path.

Specifying Profiling Summary File

The `-prof_file`*`filename`* option specifies file name for profiling summary file.

Guidelines for Using Advanced PGO

When you use PGO, consider the following guidelines:

- Minimize the changes to your program after instrumented execution and before feedback compilation. During feedback compilation, the compiler ignores dynamic information for functions modified after that information was generated.

  Note
  The compiler issues a warning that the dynamic information does not correspond to a modified function.

- Repeat the instrumentation compilation if you make many changes to your source files after execution and before feedback compilation.
- Specify the name of the profile summary file using the `-prof_file`*`filename`* option

See PGO Environment Variables.

PGO Environment Variables

The environment variables determine the directory in which to store dynamic information files or whether to overwrite `pgopti.dpi`. Refer to your operating system documentation for instructions on how to specify environment variables and their values.

The PGO environment variables are described in the table below.

Variable	Description
PROF_DIR	Specifies the directory in which dynamic information files are created. This variable applies to all three phases of the profiling process.
PROF_DUMP_INTERVAL	Initiates interval profile dumping in an instrumented user application.
PROF_NO_CLOBBER	Alters the feedback compilation phase slightly. By default, during the feedback compilation phase, the compiler merges the data from all dynamic information files and creates a new `pgopti.dpi` file, even if one already exists. When this variable is set, the compiler does not overwrite the existing `pgopti.dpi` file. Instead, the compiler issues a warning and you must remove the `pgopti.dpi` file if you want to use additional dynamic information files.

See also the documentation for your operating system for instructions on how to specify environment variables.

Example of Profile-Guided Optimization

The following is an example of the basic PGO phases:

1. **Instrumentation Compilation and Linking**—Use `-prof_gen` to produce an executable with instrumented information. Use also the `-prof_dir` option as recommended for most programs, especially if the application includes the source files located in multiple directories. `-prof_dir` ensures that the profile information is generated in one consistent place. For example:

   IA-32 applications:

   ```
   prompt>ifc -prof_gen -prof_dir/usr/profdata -c a1.f a2.f a3.f

   prompt>ifc a1.o a2.o a3.o
   ```

   Itanium®-based applications:

   ```
   prompt>efc -prof_gen -prof_dir/usr/profdata -c a1.f a2.f a3.f

   prompt>efc a1.o a2.o a3.o
   ```

   In place of the second command, you could use the linker (`ld`) directly to produce the instrumented program. If you do this, make sure you link with the `libirc.a` library.

2. **Instrumented Execution**—Run your instrumented program with a representative set of data to create a dynamic information file.

data to create a dynamic information file.

```
prompt>a1
```

The resulting dynamic information file has a unique name and `.dyn` suffix every time you run `a1`. The instrumented file helps predict how the program runs with a particular set of data. You can run the program more than once with different input data.

3. **Feedback Compilation**—Compile and link the source files with `-prof_use` to use the dynamic information to optimize your program according to its profile:

IA-32 applications:

```
prompt>ifc -prof_use -ipo a1.f a2.f a3.f
```

Itanium-based applications:

```
prompt>efc -prof_use -ipo a1.f a2.f a3.f
```

Besides the optimization, the compiler produces a `pgopti.dpi` file. You typically specify the default optimizations (`-O2`) for phase 1, and specify more advanced optimizations (`-ip` or `-ipo`) for phase 3. This example used `-O2` in phase 1 and the `-ipo` in phase 3.

Note
The compiler ignores the `-ip` or the `-ipo` options with `-prof_gen`.

See Basic PGO Options.

Merging the .dyn Files

To merge the `.dyn` files, use the `profmerge` utility.

The `profmerge` Utility

The compiler executes `profmerge` automatically during the feedback compilation phase when you specify `-prof_use`.

The command-line usage for `profmerge` is as follows:

IA-32 applications:

```
prompt>profmerge [-nologo] [-prof_dirdirname]
```

Itanium®-based applications:

```
prompt>profmerge -em -p64 [-nologo] [-prof_dirdirname]
```

where **`-prof_dirdirname`** is a `profmerge` utility option

where **/prof_dir***dirname* is a profmerge utility option.

This merges all `.dyn` files in the current directory or the directory specified by `-prof_dir`, and produces the summary file `pgopti.dpi`.

The `-prof_file`*`filename`* option enables you to specify the name of the `.dpi` file.

The command-line usage for `profmerge` with `-prof_file`*`filename`* is as follows:

IA-32 applications:

prompt>**profmerge [-nologo] [-prof_file*filename*]**

Itanium -based applications:

prompt>**profmerge -em -p64 [-nologo] [-prof_file*filename*]**

where **/prof_file*filename*** is a `profmerge` utility option.

Dumping Profile Data

This subsection provides an example of how to call the C PGO API routines from Fortran. For complete description of the PGO API support routines, see PGO API: Profile Information Generation Support.

As part of the instrumented execution phase of profile-guided optimization, the instrumented program writes profile data to the dynamic information file (`.dyn` file). The file is written after the instrumented program returns normally from `main()` or calls the standard exit function. Programs that do not terminate normally, can use the `_PGOPTI_Prof_Dump` function. During the instrumentation compilation (`-prof_gen`) you can add a call to this function to your program. Here is an example:

```
INTERFACE
SUBROUTINE PGOPTI_PROF_DUMP()
!MS$ATTRIBUTES
C,ALIAS:'PGOPTI_Prof_Dump'::PGOPTI_PROF_DUMP
END SUBROUTINE
END INTERFACE
CALL PGOPTI_PROF_DUMP()
```

Note

You must remove the call or comment it out prior to the feedback compilation with `-prof_use`.

Using profmerge to Relocate the Source Files

The compiler uses the full path to the source file for each routine to look up the profile summary information associated with that routine. By default, this prevents you from:

- Using the profile summary file (`.dpi`) if you move your application sources.
- Sharing the profile summary file with another user who is building identical application sources that are located in a different directory.

Source Relocation

To enable the movement of application sources, as well as the sharing of profile summary files, use the `profmerge` with `-src_old` and `-src_new` options. For example:

IA-32 Compiler:
```
prompt>profmerge -prof_dir c:/work -src_old c:/work/sources -
src_new d:/project/src
```

Itanium Compiler:
```
prompt>profmerge -em -p64 -prof_dir c:/work
-src_old c:/work/sources -src_new d:/project/src
```

The above command will read the `c:/work/pgopti.dpi` file. For each routine represented in the `pgopti.dpi` file, whose source path begins with the `c:/work/sources` prefix, `profmerge` replaces that prefix with `d:/project/src`. The `c:/work/pgopti.dpi` file is updated with the new source path information.

Notes

- You can execute profmerge more than once on a given pgopti.dpi file. You may need to do this if the source files are located in multiple directories. For example:

  ```
  profmerge -src_old "c:/program files" -src_new "e:/program
  files"

  profmerge -src_old c:/proj/application -src_new d:/app
  ```

- In the values specified for `-src_old` and `-src_new`, uppercase and lowercase characters are treated as identical. Likewise, forward slash (`/`) and backward slash (\) characters are treated as identical.
- Because the source relocation feature of `profmerge` modifies the `pgopti.dpi` file, you may wish to make a backup copy of the file prior to performing the source relocation.

PGO API Support

The Profile Information Generation Support (Profile IGS) enables you to control the generation of profile information during the instrumented execution phase of profile-guided optimizations.

Normally, profile information is generated by an instrumented application when it terminates by calling the standard `exit()` function.

To ensure that profile information is generated, the functions described in this section may be necessary or useful in the following situations:

- The instrumented application exits using a non-standard exit routine.
- The instrumented application is a non-terminating application: `exit()` is never called.
- The application requires control of when the profile information is generated.

A set of functions and an environment variable comprise the Profile IGS.

The Profile IGS Functions

The Profile IGS functions are available to your application by inserting a header file at the top of any source file where the functions may be used.

```
#include "pgouser.h"
```

Note

The Profile IGS functions are written in C language. Fortran applications need to call C functions.

The rest of the topics in this section describe the Profile IGS functions.

Note

Without instrumentation, the Profile IGS functions cannot provide PGO API support.

The Profile IGS Environment Variable

The environment variable for Profile IGS is `PROF_DUMP_INTERVAL`. This environment variable may be used to initiate Interval Profile Dumping in an instrumented user application. See the recommended usage of `_PGOPTI_Set_Interval_Prof_Dump()` for more information.

Dumping Profile Information

The `_PGOPTI_Prof_Dump()` function dumps the profile information collected by the instrumented application and has the following prototype:

```
void _PGOPTI_Prof_Dump(void);
```

The profile information is generated in a `.dyn` file (generated in phase 2 of the PGO).

Recommended usage

Insert a single call to this function in the body of the function which terminates the user application. Normally, `_PGOPTI_Prof_Dump()` should be called just once.

It is also possible to use this function in conjunction with the `_PGOPTI_Prof_Reset()` function to generate multiple `.dyn` files (presumably from multiple sets of input data).

Example

```
/* selectively collect profile
information
for the portion of the application
involved in processing input data
*/
input_data = get_input_data();
while (input_data) {
_PGOPTI_Prof_Reset();
process_data(input_data);
_PGOPTI_Prof_Dump();
input_data = get_input_data();
}
```

Resetting the Dynamic Profile Counters

The `_PGOPTI_Prof_Reset()` function resets the dynamic profile counters and has the following prototype:

```
void _PGOPTI_Prof_Reset(void);
```

Recommended usage

Use this function to clear the profile counters prior to collecting profile information on a section of the instrumented application. See the example under `_PGOPTI_Prof_Dump()`.

Dumping and Resetting Profile Information

The `_PGOPTI_Prof_Dump_And_Reset()` function dumps the profile information to a new `.dyn` file and then resets the dynamic profile counters. Then the execution of the instrumented application continues. The prototype of this function is:

```
void _PGOPTI_Prof_Dump_And_Reset(void);
```

This function is used in non-terminating applications and may be called more than once.

Recommended usage

Periodic calls to this function enables a non-terminating application to generate one or more profile information files (`.dyn` files). These files are merged during the feedback phase (phase 3) of profile-guided optimizations. The direct use of this function enables your

application to control precisely when the profile information is generated.

Interval Profile Dumping

The `_PGOPTI_Set_Interval_Prof_Dump()` function activates Interval Profile Dumping and sets the approximate frequency at which dumps occur. The prototype of the function call is:

```
void _PGOPTI_Set_Interval_Prof_Dump(int interval);
```

This function is used in non-terminating applications.

The *`interval`* parameter specifies the time interval at which profile dumping occurs and is measured in milliseconds. For example, if interval is set to 5000, then a profile dump and reset will occur approximately every 5 seconds. The interval is approximate because the time-check controlling the dump and reset is only performed upon entry to any instrumented function in your application.

Note

1. Setting interval to zero or a negative number will disable interval profile dumping.

2. Setting a very small value for interval may cause the instrumented application to spend nearly all of its time dumping profile information. Be sure to set interval to a large enough value so that the application can perform actual work and substantial profile information is collected.

Recommended usage

This function may be called at the start of a non-terminating user application, to initiate Interval Profile Dumping. Note that an alternative method of initiating Interval Profile Dumping is by setting the environment variable, `PROF_DUMP_INTERVAL`, to the desired interval value prior to starting the application.

The intention of Interval Profile Dumping is to allow a non-terminating application to be profiled with minimal changes to the application source code.

High-level Language Optimizations (HLO)

High-level optimizations exploit the properties of source code constructs (for example, loops and arrays) in the applications developed in high-level programming languages, such as Fortran and C++. The high-level optimizations include loop interchange, loop fusion, loop unrolling, loop distribution, unroll-and-jam, blocking, data prefetch, scalar replacement, data layout optimizations and loop unrolling techniques.

The option that turns on the high-level optimizations is `-O3`. See high-level language options summary. The scope of optimizations turned on by `-O3` is different for IA-32 and

Itanium®-based applications. See Setting Optimization Levels.

IA-32 and Itanium®-based applications	
-O3	Enable -O2 option plus more aggressive optimizations, for example, loop transformation and prefetching. -O3 optimizes for maximum speed, but may not improve performance for some programs.
IA-32 applications	
-O3	In addition, in conjunction with the vectorization options, -ax{M|K|W} and -x{M|K|W}, -O3 causes the compiler to perform more aggressive data dependency analysis than for -O2. This may result in longer compilation times.

Loop Transformations

All these transformations are supported by data dependence. These techniques also include induction variable elimination, constant propagation, copy propagation, forward substitution, and dead code elimination. The loop transformation techniques include:

- loop normalization
- loop reversal
- loop interchange and permutation
- loop skewing
- loop distribution
- loop fusion
- scalar replacement

These techniques also include induction variable elimination, constant propagation, copy propagation, forward substitution, and dead code elimination. In addition to the loop transformations listed for both IA-32 and Itanium® architectures above, the Itanium architecture enables to implement collapsing techniques.

Scalar Replacement (IA-32 Only)

The goal of scalar replacement is to reduce memory references. This is done mainly by replacing array references with register references.

While the compiler replaces some array references with register references when -O1 or -O2 is specified, more aggressive replacement is performed when -O3 (-s alar rep) is

-O2 is specified, more aggressive replacement is performed when -O3 (-scalar_rep) is specified. For example, with -O3 the compiler attempts replacement when there are loop-carried dependences or when data-dependence analysis is required for memory disambiguation.

-scalar_rep[-]	Enables (default) or disables scalar replacement performed during loop transformations (requires -O3).

Loop Unrolling with -unroll[n]

The -unroll[n] option is used in the following way:

- -unroll*n* specifies the maximum number of times you want to unroll a loop. The following example unrolls a loop at most four times:

prompt>**ifc -unroll4 *a.f***

To disable loop unrolling, specify *n* as 0. The following example disables loop unrolling:

prompt>**ifc -unroll0 *a.f***

- -unroll (*n* omitted) lets the compiler decide whether to perform unrolling or not.
- -unroll0 (*n* = 0) disables unroller.

Itanium® compiler currently uses only *n* = 0; any other value is NOP.

Benefits and Limitations of Loop Unrolling

The benefits are:

- Unrolling eliminates branches and some of the code.
- Unrolling enables you to aggressively schedule (or pipeline) the loop to hide latencies if you have enough free registers to keep variables live.
- The Pentium® 4 or Xeon (TM) processors can correctly predict the exit branch for an inner loop that has 16 or fewer iterations, if that number of iterations is predictable and there are no conditional branches in the loop. Therefore, if the loop body size is not excessive, and the probable number of iterations is known, unroll inner loops for:
  - Pentium 4 or Xeon processor, until they have a maximum of 16 iterations
  - Pentium III or Pentium II processors, until they have a maximum of 4 iterations

The potential costs are:

- Excessive unrolling, or unrolling of very large loops can lead to increased code size.
- If the number of iterations of the unrolled loop is 16 or less, the branch predictor should be able to correctly predict branches in the loop body that alternate direction.

For more information on how to optimize with `-unroll[n]`, refer to *Intel® Pentium® 4 and Intel® Xeon(TM) Processor Optimization Reference Manual*.

Memory Dependency with IVDEP Directive

The `-ivdep_parallel` option discussed below is used for Itanium®-based applications only.

The `-ivdep_parallel` option indicates there is absolutely no loop-carried memory dependency in the loop where `IVDEP` directive is specified. This technique is useful for some sparse matrix applications.

For example, the following loop requires `-ivdep_parallel` in addition to the directive `IVDEP` to indicate there is no loop-carried dependencies.

```
!DIR$IVDEP
do i=1,n
e(ix(2,i))=e(ix(2,i))+1.0
e(ix(3,i))=e(ix(3,i))+2.0
enddo
```

The following example shows that using this option and the `IVDEP` directive ensures there is no loop-carried dependency for the store into `a()`.

```
!DIR$IVDEP
do i=1,n
a(b(j)) = a(b(j))+1
enddo
```

See `IVDEP` directive for IA-32 applications.

Prefetching

The goal of `-prefetch` insertion is to reduce cache misses by providing hints to the processor about when data should be loaded into the cache. The prefetching optimizations implement the following options:

`-prefetch[-]`	Enable or disable (`-prefetch-`) prefetch insertion. This option requires that `-O3` be specified. The default with `-O3` is `-prefetch`.

To facilitate compiler optimization:

- Minimize use of global variables and pointers.
- Minimize use of complex control flow.
- Use the `const` modifier, avoid `register` modifier.
- Choose data types carefully and avoid type casting.

For more information on how to optimize with `-prefetch[-]`, refer to *Intel® Pentium® 4 and Intel® Xeon(TM) Processor Optimization Reference Manual*.

Parallelization

For shared memory parallel programming, the Intel® Fortran Compiler supports both the OpenMP* API and an automatic parallelization capability.

The compiler supports the OpenMP Fortran version 2.0 API specification and provides symmetric multiprocessing (SMP), which relieves the user from having to deal with the low-level details of iteration space partitioning, data sharing, and thread scheduling and synchronization; it also provides the performance gain from shared memory, multiprocessor systems.

The auto-parallelization feature of the Intel Fortran Compiler automatically translates serial portions of the input program into equivalent multithreaded code. Automatic parallelization determines the loops that are good worksharing candidates, performs the dataflow analysis to verify correct parallel execution, and partitions the data for threaded code generation as is needed in programming with OpenMP directives.

The following table lists the options that perform OpenMP and auto-parallelization support.

Option	Description
`-openmp`	Enables the parallelizer to generate multithreaded code based on the OpenMP directives. Default: OFF.
`-openmp_report{0|1|2}`	Controls the OpenMP parallelizer's diagnostic levels. Default: `-openmp_report1`.
`-openmp_stubs`	Enables compilation of OpenMP programs in sequential mode. The OpenMP directives are ignored and a stub OpenMP library is linked. Default: OFF.
`-parallel`	Enables the auto-parallelizer to generate multithreaded code for loops that can be safely executed in parallel. Default: OFF.
`-par_threshold{n}`	Sets a threshold for the auto-

	parallelization of loops based on the probability of profitable execution of the loop in parallel, n=0 to 100. n=0 implies "always." Default: n=75.
`-par_report{0|1|2|3}`	Controls the auto-parallelizer's diagnostic levels. Default: `-par_report1`.

Note

When both `-openmp` and `-parallel` are specified on the command line, the `-parallel` option is only honored in routines that do not contain OpenMP directives. For routines that contain OpenMP directives, only the `-openmp` option is honored.

Important component of the parallelization programming is the Intel Fortran Compiler's vectorizer. The vectorizer detects operations in the program that can be done in parallel, and then converts the sequential program to process 2, 4, 8 or up to 16 elements in one operation, depending on the data type. In some cases auto-parallelization and vectorization can be combined for better performance results.

Parallelization with OpenMP*

The Intel® Fortran Compiler supports the OpenMP* Fortran version 2.0 API specification. OpenMP provides symmetric multiprocessing (SMP) with the following major features:

- Relieves the user from having to deal with the low-level details of iteration space partitioning, data sharing, and thread scheduling and synchronization.
- Provides the benefit of the performance available from shared memory, multiprocessor systems.

The Intel Fortran Compiler performs transformations to generate multithreaded code based on the user's placement of OpenMP directives in the source program making it easy to add threading to existing software. The Intel compiler supports all of the current industry-standard OpenMP directives, except `workshare`, and compiles parallel programs annotated with OpenMP directives.

In addition, the Intel Fortran Compiler provides Intel-specific extensions to the OpenMP Fortran version 2.0 specification including runtime library routines and environment variables.

Note

As with many advanced features of compilers, you must properly understand the functionality of the OpenMP directives in order to use them effectively and avoid unwanted program behavior.

See parallelization options summary for all options of the OpenMP feature in the Intel

Fortran Compiler. For complete information on the OpenMP standard, visit the www.openmp.org web site. For complete Fortran language specifications, see the OpenMP Fortran version 2.0 specifications.

Parallel Processing with OpenMP

To compile with OpenMP, you need to prepare your program by annotating the code with OpenMP directives in the form of the Fortran program comments. The Intel Fortran Compiler first processes the application and produces a multithreaded version of the code which is then compiled. The output is a Fortran executable with the parallelism implemented by threads that execute parallel regions or constructs. See Programming with OpenMP.

Performance Analysis

For performance analysis of your program, you can use the VTune(TM) analyzer to show performance information. You can obtain detailed information about which portions of the code that require the largest amount of time to execute and where parallel performance problems are located.

Programming with OpenMP

The Intel® Fortran Compiler accepts a Fortran program containing OpenMP directives as input and produces a multithreaded version of the code. When the parallel program begins execution, a single thread exists. This thread is called the master thread. The master thread will continue to process serially until it encounters a parallel region.

Parallel Region and Constructs

A parallel region is a block of code that must be executed by a team of threads in parallel. In the OpenMP Fortran API, a parallel construct is defined by placing OpenMP directives `parallel` at the beginning and `end parallel` at the end of the code segment. Code segments thus bounded can be executed in parallel.

A structured block of code is a collection of one or more executable statements with a single point of entry at the top and a single point of exit at the bottom.

The Intel Fortran Compiler supports worksharing and synchronization constructs. Each of these constructs consists of one or two specific OpenMP directives and sometimes the enclosed or following structured block of code. For complete definitions of constructs, see the OpenMP Fortran version 2.0 specifications.

At the end of the parallel region, threads wait until all team members have arrived. The team is logically disbanded (but may be reused in the next parallel region), and the master thread continues serial execution until it encounters the next parallel region.

Worksharing Construct

A worksharing construct divides the execution of the enclosed code region among the members of the team created on entering the enclosing parallel region. When the master thread enters a parallel region, a team of threads is formed. Starting from the beginning of the parallel region, code is replicated (executed by all team members) until a `worksharing` construct is encountered. A worksharing construct divides the execution of the enclosed code among the members of the team that encounter it.

The OpenMP `sections` or `do` constructs are defined as `worksharing` constructs because they distribute the enclosed work among the threads of the current team. A `worksharing` construct is only distributed if it is encountered during dynamic execution of a parallel region. If the `worksharing` construct occurs lexically inside of the parallel region, then it is always executed by distributing the work among the team members. If the `worksharing` construct is not lexically (explicitly) enclosed by a parallel region (that is, it is `orphaned`), then the `worksharing` construct will be distributed among the team members of the closest dynamically-enclosing parallel region, if one exists. Otherwise, it will be executed serially.

When a thread reaches the end of a `worksharing` construct, it may wait until all team members within that construct have completed their work. When all of the work defined by the worksharing construct is finished, the team exits the worksharing construct and continues executing the code that follows.

Parallel Processing Directive Groups

The parallel processing directives include the following groups:

Worksharing

- The `do` and `end do` directives specify parallel execution of loop iterations.
- The `sections` directive specifies parallel execution for arbitrary blocks of sequential code. Each `section` is executed once by a thread in the team.
- The `single` directive defines a section of code where exactly one thread is allowed to execute the code; threads not chosen to execute this section ignore the code.

Synchronization and `master`

Synchronization is the interthread communication that ensures the consistency of shared data and coordinates parallel execution among threads. Shared data is consistent within a team of threads when all threads obtain the identical value when the data is accessed. A synchronization construct is used to insure this consistency of the shared data.

- The OpenMP synchronization directives are `critical`, `ordered`, `atomic`, `flush`, and `barrier`.
- Within a parallel region or a `worksharing` construct only one thread at a time is

- Within a parallel region or a workshare construct only one thread at a time is allowed to execute the code within a `critical` construct.
- The `ordered` directive is used in conjunction with a `do` or `sections` construct to impose a serial order on the execution of a section of code.
- The `atomic` directive is used to update a memory location in an uninterruptable fashion.
- The `flush` directive is used to insure that all threads in a team have a consistent view of memory.
- A `barrier` directive forces all team members to gather at a particular point in code. Each team member that executes a `barrier` waits at the `barrier` until all of the team members have arrived. A `barrier` cannot be used within `worksharing` or other synchronization constructs due to the potential for deadlock.
- The `master` directive is used to force execution by the master thread.

See the list of OpenMP Directives and Clauses.

Data Sharing

Data sharing is specified at the start of a parallel region or `worksharing` construct by using the `shared` and `private` clauses. All variables in the `shared` clause are shared among the members of a team. It is the application's responsibility to:

- synchronize access to these variables. All variables in the `private` clause are private to each team member. For the entire parallel region, assuming *t* team members, there are *t*+1 copies of all the variables in the `private` clause: one global copy that is active outside parallel regions and a `private` copy for each team member.
- initialize `private` variables at the start of a parallel region, unless the `firstprivate` clause is specified. In this case, the `private` copy is initialized from the global copy at the start of the construct at which the `firstprivate` clause is specified.
- update the global copy of a `private` variable at the end of a parallel region. However, the `lastprivate` clause of a `DO` directive enables updating the global copy from the team member that executed serially the last iteration of the loop.

In addition to `shared` and `private` variables, individual variables and entire `common` blocks can be privatized using the `threadprivate` directive.

Orphaned Directives

OpenMP contains a feature called orphaning which dramatically increases the expressiveness of parallel directives. Orphaning is a situation when directives related to a parallel region are not required to occur lexically within a single program unit. Directives

parallel region are not required to occur lexically within a single program unit. Directives such as `critical`, `barrier`, `sections`, `single`, `master`, and `do`, can occur by themselves in a program unit, dynamically “binding” to the enclosing parallel region at run time.

Orphaned directives enable parallelism to be inserted into existing code with a minimum of code restructuring. Orphaning can also improve performance by enabling a single parallel region to bind with multiple `do` directives located within called subroutines. Consider the following code segment:

```
...
!$omp parallel
call phase1
call phase2
!$omp end parallel
...

subroutine phase1
!$omp do private(i) shared(n)
do i = 1, n
call some_work(i)
end do
!$omp end do
end

subroutine phase2
!$omp do private(j) shared(n)
do j = 1, n
call more_work(j)
end do
!$omp end do
end
```

Orphaned Directives Usage Rules

- An orphaned `worksharing` construct (`section`, `single`, `do`) is executed by a team consisting of one thread, that is, serially.
- Any collective operation (`worksharing` construct or `barrier`) executed inside of a `worksharing` construct is illegal.
- It is illegal to execute a collective operation (`worksharing` construct or `barrier`) from within a synchronization region (`critical`/`ordered`).
- The opening and closing directives of a directive pair (for example, `do` - `end do`) must occur in a single block of the program.
- Private scoping of a variable can be specified at a `worksharing` construct. Shared scoping must be specified at the parallel region. For complete details, see the OpenMP Fortran version 2.0 specifications.

Preparing Code for OpenMP Processing

The following are the major stages and steps of preparing your code for using OpenMP. Typically, the first two stages can be done on uniprocessor or multiprocessor systems; later stages are typically done only on multiprocessor systems.

Before Inserting OpenMP Directives

Before inserting any OpenMP parallel directives, verify that your code is safe for parallel execution by doing the following:

- Place local variables on the stack. This is the default behavior of the Intel Fortran Compiler when `-openmp` is used.
- Use `-auto` or similar (`-auto_scalar`) compiler option to make the locals automatic. Avoid using compiler options that inhibit stack allocation of local variables. By default (`-auto_scalar`) local scalar variables become shared across threads, so you may need to add synchronization code to ensure proper access by threads.

Analyze

The analysis includes the following major actions:

- Profile the program to find out where it spends most of its time. This is the part of the program that benefits most from parallelization efforts. This stage can be accomplished using basic PGO options.
- Wherever the program contains nested loops, choose the outer-most loop, which has very few cross-iteration dependencies.

Restructure

- To restructure your program for successful OpenMP implementation, you can perform some or all of the following actions:

1. If a chosen loop is able to execute iterations in parallel, introduce a `parallel do` construct around this loop.
2. Try to remove any cross-iteration dependencies by rewriting the algorithm.
3. Synchronize the remaining cross-iteration dependencies by placing `critical` constructs around the uses and assignments to variables involved in the dependencies.
4. List the variables that are present in the loop within appropriate `shared`, `private`, `lastprivate`, `firstprivate`, or `reduction` clauses.
5. List the `do` index of the parallel loop as `private`. This step is optional.

6. `common` block elements must not be placed on the `private` list if their global scope is to be preserved. The `threadprivate` directive can be used to privatize to each thread the `common` block containing those variables with global scope. `threadprivate` creates a copy of the `common` block for each of the threads in the team.

7. Any I/O in the parallel region should be synchronized.

8. Identify more parallel loops and restructure them.

9. If possible, merge adjacent `parallel do` constructs into a single parallel region containing multiple `do` directives to reduce execution overhead.

Tune

The tuning process should include minimizing the sequential code in critical sections and load balancing by using the `schedule` clause or the `omp_schedule` environment variable.

Note

This step is typically performed on a multiprocessor system.

Parallel Processing Thread Model

This topic explains the processing of the parallelized program and adds more definitions of the terms used in the parallel programming.

The Execution Flow

As mentioned in previous topic, a program containing OpenMP Fortran API compiler directives begins execution as a single process, called the **master** thread of execution. The master thread executes sequentially until the first **parallel construct** is encountered.

In OpenMP Fortran API, the `PARALLEL` and `END PARALLEL` directives define the parallel construct. When the master thread encounters a parallel construct, it creates a **team** of threads, with the master thread becoming the master of the team. The program statements enclosed by the parallel construct are executed in parallel by each thread in the team. These statements include routines called from within the enclosed statements.

The statements enclosed lexically within a construct define the **static extent** of the construct. The **dynamic extent** includes the static extent as well as the routines called from within the construct. When the `END PARALLEL` directive is encountered, the threads in the team synchronize at that point, the team is dissolved, and only the master thread continues execution. The other threads in the team enter a wait state.

You can specify any number of parallel constructs in a single program. As a result, thread teams can be created and dissolved many times during program execution.

Using Orphaned Directives

In routines called from within parallel constructs, you can also use directives. Directives that are not in the lexical extent of the parallel construct, but are in the dynamic extent, are called orphaned directives. Orphaned directives allow you to execute major portions of your program in parallel with only minimal changes to the sequential version of the program. Using this functionality, you can code parallel constructs at the top levels of your program call tree and use directives to control execution in any of the called routines. For example:

```
subroutine F
...
!$OMP parallel...
...
    call G
...
subroutine G
...
!$OMP DO...
...
```

The `!$OMP DO` is an orphaned directive because the parallel region it will execute in is not lexically present in G.

Data Environment Directive

A data environment directive controls the data environment during the execution of parallel constructs.

You can control the data environment within parallel and worksharing constructs. Using directives and data environment clauses on directives, you can:

- Privatize named common blocks by using `THREADPRIVATE` directive
- Control data scope attributes by using the `THREADPRIVATE` directive's clauses.

  The data scope attribute clauses are:

  - COPYIN
  - DEFAULT
  - PRIVATE
  - FIRSTPRIVATE
  - LASTPRIVATE
  - REDUCTION

- SHARED

You can use several directive clauses to control the data scope attributes of variables for the duration of the construct in which you specify them. If you do not specify a data scope attribute clause on a directive, the default is SHARED for those variables affected by the directive.

For detailed descriptions of the clauses, see the OpenMP Fortran version 2.0 specifications.

Pseudo Code of the Parallel Processing Model

A sample program using some of the more common OpenMP directives is shown in the code example that follows. This example also indicates the difference between serial regions and parallel regions.

```
program main          ! Begin Serial Execution
...                   ! Only the master thread executes
!$omp parallel       ! Begin a Parallel Construct, form a
                      team
...                   ! This is Replicated Code where each
                      team ! member executes the same code
!$omp sections       ! Begin a Worksharing Construct
!$omp section        ! One unit of work
...                   !
!$omp section        ! Another unit of work
...                   !
!$omp end            ! Wait until both units of work
sections              complete
...                   ! More Replicated Code
  !$omp do            ! Begin a Worksharing Construct,
   do                 ! each iteration is a unit of work
   ...                ! Work is distributed among the team
   end do             !
!$omp end do         ! End of Worksharing Construct,
nowait                nowait is
                      ! specified
 ...                  ! More Replicated Code
!$omp end            ! End of Parallel Construct, disband
parallel              team ! and continue with serial
                      execution
...                   ! Possibly more Parallel Constructs
end                   ! End serial execution
```

Compiling with OpenMP, Directive Format, and Diagnostics

To run the Intel® Fortran Compiler in OpenMP mode, you need to invoke the Intel compiler with the
`-openmp` option:

IA-32 applications:

`ifc -openmp` *`input_file(s)`*

Itanium®-based applications:

`efc -openmp` *`input_file(s)`*

Before you run the multithreaded code, you can set the number of desired threads to the OpenMP environment variable, `OMP_NUM_THREADS`. See the OpenMP Environment Variables section for further information. The Intel Extensjon Routines topic describes the OpenMP extensions to the specification that have been added by Intel in the Intel® Fortran Compiler.

`-openmp` Option

The `-openmp` option enables the parallelizer to generate multithreaded code based on the OpenMP directives. The code can be executed in parallel on both uniprocessor and multiprocessor systems.

The `-openmp` option works with both `-O0` (no optimization) and any optimization level of `-O1`,
`-O2` (default) and `-O3`. Specifying `-O0` with `-openmp` helps to debug OpenMP applications.

When you use the `-openmp` option, the compiler sets the `-auto` option (causes all variables to be allocated on the stack, rather than in local static storage.) for the compiler unless you specified it on the command line.

OpenMP Directive Format and Syntax

The OpenMP directives use the following format:

```
<prefix> <directive> [<clause> [[,] <clause> . . .]]
```

where the brackets above mean:

- `<xxx>:` the prefix and directive are required
- `[<xxx>]:` if a directive uses one clause or more, the clause(s) is required
- `[,]:` commas between the `<clause>`s are optional

- [,]: commas between the clauses are optional.

For fixed form source input, the prefix is `!$omp` or `c$omp`

For free form source input, the prefix is `!$omp` only.

The prefix is followed by the directive name; for example:

```
!$omp parallel
```

Since OpenMP directives begin with an exclamation point, the directives take the form of comments if you omit the `-openmp` option.

Syntax for Parallel Regions in the Source Code

The OpenMP constructs defining a parallel region have one of the following syntax forms:

```
!$omp <directive>

<structured block of code>

!$omp end <directive>
```

or

```
!$omp <directive>

<structured block of code>
```

or

```
!$omp <directive>
```

where `<directive>` is the name of a particular OpenMP directive.

OpenMP Diagnostics

The `-openmp_report{0|1|2}` option controls the OpenMP parallelizer's diagnostic levels `0`, `1`, or `2` as follows:

`-openmp_report0` = no diagnostic information is displayed.

`-openmp_report1` = display diagnostics indicating loops, regions, and sections successfully parallelized.

`-openmp_report2` = same as `-openmp_report1` plus diagnostics indicating `master` constructs, `single` constructs, `critical` constructs, `ordered` constructs, `atomic` directives, etc. successfully handled.

The default is `-openmp_report1`.

OpenMP Directives and Clauses

This topic provides a summary of the OpenMP directives and clauses. For detailed descriptions, see the OpenMP Fortran version 2.0 specifications.

OpenMP Directives

Directive	Description
`parallel` `end parallel`	Defines a parallel region.
`do` `end do`	Identifies an iterative `worksharing` construct in which the iterations of the associated loop should be executed in parallel.
`sections` `end sections`	Identifies a non-iterative `worksharing` construct that specifies a set of structured blocks that are to be divided among threads in a team.
`section`	Indicates that the associated structured block should be executed in parallel as part of the enclosing sections construct.
`single` `end single`	Identifies a construct that specifies that the associated structured block is executed by only one thread in the team.
`parallel do` `end parallel do`	A shortcut for a `parallel` region that contains a single `do` directive. **Note** The `parallel do` or `do` OpenMP directive must be immediately followed by a `do` statement (`do-stmt` as defined by R818 of the ANSI Fortran standard). If you place another statement or an OpenMP directive between the `parallel do` or `do` directive and the `do` statement, the Intel Fortran Compiler issues a syntax error.

`parallel`
`sections`
`end parallel`
`sections`
	Provides a shortcut form for specifying a parallel region containing a single `sections` construct.

`master` `end master`	Identifies a construct that specifies a structured block that is executed by only the `master` thread of the team.

`critical[`*`lock`*`]`
`end critical`
`[`*`lock`*`]`
	Identifies a construct that restricts execution of the associated structured block to a single thread at a time. Each thread waits at the beginning of the critical construct until no other thread is executing a critical

	construct until no other thread is executing a critical construct with the same *lock* argument.
barrier	Synchronizes all the threads in a team. Each thread waits until all of the other threads in that team have reached this point.
atomic	Ensures that a specific memory location is updated atomically, rather than exposing it to the possibility of multiple, simultaneously writing threads.
flush [(*list*)]	Specifies a "cross-thread" sequence point at which the implementation is required to ensure that all the threads in a team have a consistent view of certain objects in memory. The optional *list* argument consists of a comma-separated list of variables to be flushed.
ordered end ordered	The structured block following an ordered directive is executed in the order in which iterations would be executed in a sequential loop.
threadprivate (*list*)	Makes the named common blocks or variables private to a thread. The *list* argument consists of a comma-separated list of common blocks or variables.

OpenMP Clauses

Clause	Description
private (*list*)	Declares variables in *list* to be private To each thread in a team.
firstprivate (*list*)	Same as private, but the copy of each variable in the *list* is initialized using the value of the original variable existing before the construct.
lastprivate (*list*)	Same as private, but the original variables in *list* are updated using the values assigned to the corresponding private variables in the last iteration in the do construct loop or the last section construct.
copyprivate (*list*)	Uses private variables in *list* to broadcast values, or pointers to shared objects, from one member of a team to the other members at the end of a single construct.
nowait	Specifies that threads need not wait at the end of worksharing constructs until they have completed execution. The threads may proceed past the end of the worksharing

	constructs as soon as there is no more work available for them to execute.
`shared (`*`list`*`)`	Shares variables in *`list`* among all the threads in a team.
`default (`*`mode`*`)`	Determines the default data-scope attributes of variables not explicitly specified by another clause. Possible values for *`mode`* are `private`, `shared`, or `none`.
`reduction ({operator|intrinsic}:`*`list`*`)`	Performs a reduction on variables that appear in *`list`* with the operator `operator` or the intrinsic procedure name `intrinsic`; `operator` is one of the following: `+`, `*`, `.and.`, `.or.`, `.eqv.`, `.neqv.`; `intrinsic` refers to one of the following: `max`, `min`, `iand`, `ior`, or `ieor`.
`ordered` `end ordered`	Used in conjunction with a `do` or `sections` construct to impose a serial order on the execution of a section of code. If `ordered` constructs are contained in the dynamic extent of the `do` construct, the ordered clause must be present on the `do` directive.
`if (`*`scalar_logical_expression`*`)`	The enclosed parallel region is executed in parallel only if the *`scalar_logical_expression`* evaluates to `.true.`; otherwise the parallel region is serialized.
`num_threads (`*`scalar_integer_expression`*`)`	Requests the number of threads specified by *`scalar_integer_expression`* for the parallel region.
`schedule (`*`type`*`[,`*`chunk`*`])`	Specifies how iterations of the `do` construct are divided among the threads of the team. Possible values for the *`type`* argument are `static`, `dynamic`, `guided`, and `runtime`. The optional *`chunk`* argument must be a positive scalar integer expression.
`copyin (`*`list`*`)`	Specifies that the master thread's data values be copied to the `threadprivate`'s copies of the common blocks or variables specified in *`list`* at the beginning of the

	in list at the beginning of the parallel region.

OpenMP Support Libraries

The Intel Fortran Compiler with OpenMP support provides a production support library, `libguide.lib`. This library enables you to run an application under different execution modes. It is used for normal or performance-critical runs on applications that have already been tuned.

Execution modes

The compiler with OpenMP enables you to run an application under different execution modes that can be specified at run time. The libraries support the serial, turnaround, and throughput modes. These modes are selected by using the `kmp_library` environment variable at run time.

Serial

The serial mode forces parallel applications to run on a single processor.

Turnaround

In a dedicated (batch or single user) parallel environment where all processors are exclusively allocated to the program for its entire run, it is most important to effectively utilize all of the processors all of the time. The turnaround mode is designed to keep active all of the processors involved in the parallel computation in order to minimize the execution time of a single job. In this mode, the worker threads actively wait for more parallel work, without yielding to other threads.

Note

Avoid over-allocating system resources. This occurs if either too many threads have been specified, or if too few processors are available at run time. If system resources are over-allocated, this mode will cause poor performance. The throughput mode should be used instead if this occurs.

Throughput

In a multi-user environment where the load on the parallel machine is not constant or where the job stream is not predictable, it may be better to design and tune for throughput. This minimizes the total time to run multiple jobs simultaneously. In this mode, the worker threads will yield to other threads while waiting for more parallel work.

The throughput mode is designed to make the program aware of its environment (that is, the system load) and to adjust its resource usage to produce efficient execution in a dynamic environment. This mode is the default.

OpenMP Environment Variables

OpenMP* Environment Variables

This topic describes the standard OpenMP environment variables (with the OMP_ prefix) and Intel-specific environment variables (with the KMP_ prefix) that are Intel extensions to the standard Fortran Compiler .

Standard Environment Variables

Variable	Description	Default
OMP_SCHEDULE	Sets the run-time schedule type and chunk size.	static, no chunk size specified
OMP_NUM_THREADS	Sets the number of threads to use during execution.	Number of processors
OMP_DYNAMIC	Enables (.true.) or disables (.false.) the dynamic adjustment of the number of threads.	.false.
OMP_NESTED	Enables (.true.) or disables (.false.)nested parallelism.	.false.

Intel Extension Environment Variables

Environment Variable	Description	Default
KMP_LIBRARY	Selects the OpenMP runtime library throughput. The options for the variable value are: serial, turnaround, or throughput indicating the execution mode. The default value of throughput is used if this variable is not specified.	throughput (execution mode)
KMP_STACKSIZE	Sets the number of bytes to allocate for each parallel thread to use as its private stack. Use the optional suffix b, k, m, g, or t, to specify bytes, kilobytes, megabytes, gigabytes, or terabytes.	IA-32: 2m Itanium compiler: 4m

OpenMP Runtime Library Routines

OpenMP* Runtime Library Routines

OpenMP provides several runtime library routines to assist you in managing your program in parallel mode. Many of these runtime library routines have corresponding environment variables that can be set as defaults. The runtime library routines enable you to dynamically change these factors to assist in controlling your program. In all cases, a call to a runtime library routine overrides any corresponding environment variable.

The following table specifies the interface to these routines. The names for the routines are in user name space. The `omp_lib.f`, `omp_lib.h` and `omp_lib.mod` header files are provided in the `include` directory of your compiler installation. The `omp_lib.h` header file is provided in the `include` directory of your compiler installation for use with the Fortran `INCLUDE` statement. The `omp_lib.mod` file is provided in the Include directory for use with the Fortran `USE` statement.

There are definitions for two different locks, `omp_lock_t` and `omp_nest_lock_t`, which are used by the functions in the table that follows.

This topic provides a summary of the OpenMP runtime library routines. For detailed descriptions, see the OpenMP Fortran version 2.0 specifications.

Function	Description
Execution Environment Routines	
`subroutine omp_set_num_threads (num_threads)` `integer num_threads`	Sets the number of threads to use for subsequent parallel regions.
`integer function omp_get_num_threads()`	Returns the number of threads that are being used in the current parallel region.
`integer function omp_get_max_threads()`	Returns the maximum number of threads that are available for parallel execution.
`integer function omp_get_thread_num()`	Determines the unique thread number of the thread currently executing this section of code.
`integer function omp_get_num_procs ()`	Determines the number of processors available to the program.
`logical function omp_in_parallel()`	Returns `.true.` if called within the dynamic extent of a parallel region executing in parallel; otherwise returns `.false.`.
`subroutine omp_set_dynamic (dynamic_threads) logical dynamic_threads`	Enables or disables dynamic adjustment of the number of threads used to execute a parallel region. If d*ynamic_threads* is `.true.`, dynamic threads are enabled. If *dynamic_threads* is `.false.`,

	dynamic threads are disabled. Dynamics threads are disabled by default.
`logicl function omp_get_dynamic()`	Returns `.true.` if dynamic thread adjustment is enabled, otherwise returns `.false.`.
`subroutine omp_set_nested(`*`nested`*`)` `integer` *`nested`*	Enables or disables nested parallelism. If *`nested`* is `.true.`, nested parallelism is enabled. If *`nested`* is `.false.`, nested parallelism is disabled. Nested parallelism is disabled by default.
`logical function omp_get_nested()`	Returns `.true.` if nested parallelism is enabled, otherwise returns `.false.`.
Lock Routines	
`subroutine omp_init_lock(`*`lock`*`)` `integer (kind=omp_lock_kind)::`*`lock`*	Initializes the lock associated with *`lock`* for use in subsequent calls.
`subroutine omp_destroy_lock(`*`lock`*`)` `integer (kind=omp_lock_kind)::`*`lock`*	Causes the lock associated with *`lock`* to become undefined.
`subroutine omp_set_lock(`*`lock`*`)` `integer (kind=omp_lock_kind)::`*`lock`*	Forces the executing thread to wait until the lock associated with *`lock`* is available. The thread is granted ownership of the lock when it becomes available.
`subroutine omp_unset_lock(`*`lock`*`)` `integer (kind=omp_lock_kind)::`*`lock`*	Releases the executing thread from ownership of the lock associated with *`lock`*. The behavior is undefined if the executing thread does not own the lock associated with *`lock`*.
`logical omp_test_lock(`*`lock`*`)` `integer (kind=omp_lock_kind)::`*`lock`*	Attempts to set the lock associated with *`lock`*. If successful, returns `.true.`, otherwise returns `.false.`.
`subroutine omp_init_nest_lock (lock)` `integer (kind=omp_nest_lock_kind)::`*`lock`*	Initializes the nested lock associated with *`lock`* for use in the subsequent calls.
`subroutine omp_destroy_nest_lock (`*`lock`*`)` `integer (kind=omp_nest_lock_kind)::`*`lock`*	Causes the nested lock associated with *`lock`* to become undefined.
`subroutine omp_set_nest_lock(lock)` `integer (kind=omp_nest_lock_kind)::`*`lock`*	Forces the executing thread to wait until the nested lock associated with *`lock`* is available. The thread is granted ownership of the nested lock when it becomes available.

	when it becomes available.
`subroutine omp_unset_nest_lock (lock)` `integer (kind=omp_nest_lock_kind)::`*`lock`*	Releases the executing thread from ownership of the nested lock associated with *`lock`* if the nesting count is zero. Behavior is undefined if the executing thread does not own the nested lock associated with *`lock`*.
`integer omp_test_nest_lock(`*`lock`*`)` `integer (kind=omp_nest_lock_kind)::`*`lock`*	Attempts to set the nested lock associated with *`lock`*. If successful, returns the nesting count, otherwise returns zero.
Timing Routines	
`double-precision function omp_get_wtime()`	Returns a double-precision value equal to the elapsed wallclock time (in seconds) relative to an arbitrary reference time. The reference time does not change during program execution.
`double-precision function omp_get_wtick()`	Returns a double-precision value equal to the number of seconds between successive clock ticks.

Intel Extension Routines

The Intel® Fortran Compiler implements the following group of routines as an extension to the OpenMP runtime library: getting and setting stack size for parallel threads and memory allocation.

The Intel extension routines described in this section can be used for low-level debugging to verify that the library code and application are functioning as intended. It is recommended to use these routines with caution because using them requires the use of the `-openmp_stubs` command-line option to execute the program sequentially. These routines are also generally not recognized by other vendor's OpenMP-compliant compilers, which may cause the link stage to fail for these other compilers.

Stack Size

In most cases, directives can be used in place of the extension library routines. For example, the stack size of the parallel threads may be set using the `KMP_STACKSIZE` environment variable rather than the `kmp_set_stacksize()` library routine.

Note

A runtime call to an Intel extension routine takes precedence over the corresponding environment variable setting.

See the definitions of stack size routines in the table that follows.

Memory Allocation

The Intel® Fortran Compiler implements a group of memory allocation routines as an extension to the OpenMP* runtime library to enable threads to allocate memory from a heap local to each thread. These routines are: `kmp_malloc`, `kmp_calloc`, and `kmp_realloc`.

The memory allocated by these routines must also be freed by the `kmp_free` routine. While it is legal for the memory to be allocated by one thread and `kmp_free`'d by a different thread, this mode of operation has a slight performance penalty.

See the definitions of these routines in the table that follows.

Function/Routine	Description
Stack Size	
`function kmp_get_stacksize_s()` `integer (kind=kmp_size_t_kind) kmp_get_stacksize_s`	Returns the number of bytes that will be allocated for each parallel thread to use as its private stack. This value can be changed via the `kmp_get_stacksize_s` routine, prior to the first parallel region or via the `KMP_STACKSIZE` environment variable.
`function kmp_get_stacksize()` `integer kmp_get_stacksize`	This routine is provided for backwards compatibility only; use `kmp_get_stacksize_s` routine for compatibility across different families of Intel processors.
`subroutine kmp_set_stacksize_s(`*`size`*`)` `integer (kind=kmp_size_t_kind)` *`size`*	Sets to *`size`* the number of bytes that will be allocated for each parallel thread to use as its private stack. This value can also be set via the `KMP_STACKSIZE` environment variable. In order for `kmp_set_stacksize_s` to have an effect, it must be called before the beginning of the first (dynamically executed) parallel region in the program.
`subroutine kmp_set_stacksize(`*`size`*`)` `integer` *`size`*	This routine is provided for backward compatibility only; use `kmp_set_stacksize_s(`*`size`*`)` for compatibility across different families of Intel processors.
Memory Allocation	

`function kmp_malloc(`*`size`*`)`
`integer (kind=kmp_pointer_kind) kmp_malloc`
`integer`
	Allocate memory block of *`size`* bytes from thread-local heap.

integer (kind=kmp_size_t_kind)*size*	
function kmp_calloc (*nelem*,*elsize*) integer (kind=kmp_pointer_kind) kmp_calloc integer (kind=kmp_size_t_kind)*nelem* integer (kind=kmp_size_t_kind)*elsize*	Allocate array of *nelem* elements of size *elsize* from thread-local heap.
function kmp_realloc(*ptr*, *size*) integer (kind=kmp_pointer_kind) kmp_realloc integer (kind=kmp_pointer_kind)*ptr* integer (kind=kmp_size_t_kind)*size*	Reallocate memory block at address *ptr* and *size* bytes from thread-local heap.
subroutine kmp_free(*ptr*) integer (kind=kmp_pointer_kind) *ptr*	Free memory block at address *ptr* from thread-local heap. Memory must have been previously allocated with kmp_malloc, kmp_calloc, or kmp_realloc.

Examples of OpenMP Usage

The following examples show how to use the OpenMP feature. See more examples in the OpenMP Fortran version 2.0 specifications.

do: A Simple Difference Operator

This example shows a simple parallel loop where each iteration contains a different number of instructions. To get good load balancing, dynamic scheduling is used. The `end do` has a `nowait` because there is an implicit `barrier` at the end of the parallel region.

```
      subroutine do_1 (a,b,n)
real a(n,n), b(n,n)
c$omp parallel
c$omp&   shared(a,b,n)
c$omp&   private(i,j)
c$omp do schedule(dynamic,1)
do i = 2, n
do j = 1, i
b(j,i) = ( a(j,i) + a(j,i-1) ) / 2
enddo
enddo
c$omp end do nowait
c$omp end parallel
end
```

do: Two Difference Operators

This example shows two parallel regions fused to reduce `fork/join` overhead. The first `end do` has a `nowait` because all the data used in the second loop is different than all the data used in the first loop.

```
       subroutine do_2 (a,b,c,d,m,n)
real a(n,n), b(n,n), c(m,m), d(m,m)
c$omp parallel
c$omp&   shared(a,b,c,d,m,n)
c$omp&   private(i,j)
c$omp do schedule(dynamic,1)
do i = 2, n
do j = 1, i
b(j,i) = ( a(j,i) + a(j,i-1) ) / 2
enddo
enddo
c$omp end do nowait
c$omp do schedule(dynamic,1)
do i = 2, m
do j = 1, i
d(j,i) = ( c(j,i) + c(j,i-1) ) / 2
enddo
enddo
c$omp end do nowait
c$omp end parallel
end
```

sections: Two Difference Operators

This example demonstrates the use of the `sections` directive. The logic is identical to the preceding do example, but uses `sections` instead of do. Here the speedup is limited to 2 because there are only two units of

work whereas in do: Two Difference Operators above there are `n-1 + m-1` units of work.

```
  subroutine sections_1
(a,b,c,d,m,n)
real a(n,n), b(n,n), c(m,m), d
(m,m)
!$omp parallel
!$omp& shared(a,b,c,d,m,n)
!$omp& private(i,j)
!$omp sections
!$omp section
do i = 2, n
do j = 1, i
b(j,i)=( a(j,i) + a(j,i-1) ) / 2
enddo
```

```
enddo

!$omp section
do i = 2, m
do j = 1, i
d(j,i)=( c(j,i) + c(j,i-1) ) / 2
enddo
enddo
!$omp end sections nowait
!$omp end parallel
end
```

`single`: Updating a Shared Scalar

This example demonstrates how to use a `single` construct to update an element of the shared array a. The optional `nowait` after the first loop is omitted because it is necessary to wait at the end of the loop before proceeding into the `single` construct.

```
  subroutine sp_1a
(a,b,n)
real a(n), b(n)
!$omp parallel
!$omp& shared(a,b,n)
!$omp& private(i)
!$omp do
do i = 1, n
a(i) = 1.0 / a(i)
enddo
!$omp single
a(1) = min( a(1), 1.0 )
!$omp end single
!$omp do
do i = 1, n
b(i) = b(i) / a(i)
enddo
!$omp end do nowait
!$omp end parallel
end
```

Auto-parallelization

The auto-parallelization feature of the Intel® Fortran Compiler automatically translates serial portions of the input program into equivalent multithreaded code. The auto-parallelizer analyzes the dataflow of the program's loops and generates multithreaded code for those loops which can be safely and efficiently executed in parallel. This enables the potential exploitation of the parallel architecture found in symmetric multiprocessor (SMP) systems.

Automatic parallelization relieves the user from:

- having to deal with the details of finding loops that are good worksharing candidates
- performing the dataflow analysis to verify correct parallel execution
- partitioning the data for threaded code generation as is needed in programming with OpenMP* directives.

The parallel runtime support provides the same runtime features as found in OpenMP, such as handling the details of loop iteration modification, thread scheduling, and synchronization.

While OpenMP directives enable serial applications to transform into parallel applications quickly, the programmer must explicitly identify specific portions of the application code that contain parallelism and add the appropriate compiler directives. Auto-parallelization triggered by the `-parallel` option automatically identifies those loop structures, which contain parallelism. During compilation, the compiler automatically attempts to decompose the code sequences into separate threads for parallel processing. No other effort by the programmer is needed.

The following example illustrates how a loop's iteration space can be divided so that it can be executed concurrently on two threads:

Original Serial Code

```
do i=1,100
   a(i) = a(i) + b(i) * c(i)
enddo
```

Transformed Parallel Code

Thread 1

`do i=1,50`
`a(i) = a(i) + b(i) * c(i)`
`enddo`

Thread 2

`do i=50,100`
`a(i) = a(i) + b(i) * c(i)`
`enddo`

Programming with Auto-parallelization

Auto-parallelization feature implements some concepts of OpenMP, such as worksharing construct (with the `PARALLEL DO` directive). See Programming with OpenMP for worksharing construct. This section provides specifics of auto-parallelization.

Guidelines for Effective Auto-parallelization Usage

A loop is parallelizable if:

- The loop is countable at compile time: this means that an expression representing how many times the loop will execute (also called "the loop trip count") can be generated just before entering the loop.
- There are no `FLOW` (`READ` after `WRITE`), `OUTPUT` (`WRITE` after `READ`) or `ANTI` (`WRITE` after `READ`) loop-carried data dependences. A loop-carried data dependence occurs when the same memory location is referenced in different iterations of the loop. At the compiler's discretion, a loop may be parallelized if any assumed inhibiting loop-carried dependencies can be resolved by runtime dependency testing.

The compiler may generate a runtime test for the profitability of executing in parallel for loop with loop parameters that are not compile-time constants.

Coding Guidelines

Enhance the power and effectiveness of the auto-parallelizer by following these coding guidelines:

- Expose the trip count of loops whenever possible; specifically use constants where the trip count is known and save loop parameters in local variables.
- Avoid placing structures inside loop bodies that the compiler may assume to carry dependent data, for example, procedure calls, ambiguous indirect references or global references.
- Insert the `!DIR$ PARALLEL` directive to disambiguate assumed data dependencies.
- Insert the `!DIR$ NOPARALLEL` directive before loops known to have insufficient work to justify the overhead of sharing among threads.

Auto-parallelization Data Flow

For auto-parallelization processing, the compiler performs the following steps:

Data flow analysis ---> Loop classification ---> Dependence analysis ---> High-level parallelization --> Data partitioning ---> Multi-threaded code generation.

These steps include:

- Data flow analysis: compute the flow of data through the program
- Loop classification: determine loop candidates for parallelization based on correctness and efficiency as shown by threshold analysis
- Dependence analysis: compute the dependence analysis for references in each loop

nest

- High-level parallelization:

  - analyze dependence graph to determine loops which can execute in parallel.

  - compute runtime dependency

- Data partitioning: examine data reference and partition based on the following types of access: `SHARED`, `PRIVATE`, and `FIRSTPRIVATE`

- Multi-threaded code generation:

  - modify loop parameters

  - generate entry/exit per threaded task

  - generate calls to parallel runtime routines for thread creation and synchronization

Programming Enabling, Options, Directives, and Environment Variables

To enable the auto-parallelizer, use the `-parallel` option. The `-parallel` option detects parallel loops capable of being executed safely in parallel and automatically generates multithreaded code for these loops. An example of the command using auto -parallelization is as follows:

IA-32 compilations:

prompt>**ifc -c -parallel *myprog.f***

Itanium®-based compilations:

prompt>**efc -c -parallel *myprog.f***

Auto-parallelization Options

The `-parallel` option enables the auto-parallelizer if the `-O2` (or `-O3`) optimization option is also on (the default is `-O2`). The `-parallel` option detects parallel loops capable of being executed safely in parallel and automatically generates multithreaded code for these loops.

`-parallel`	Enables the auto-parallelizer
`-parallel_threshold{1-100}`	Controls the work threshold needed for auto-parallelization, see later subsection.
`-par_report{1|2|3}`	Controls the diagnostic messages from the auto-parallelizer, see

	from the auto-parallelizer, see later subsection.

Auto-parallelization Directives

Auto-parallelization uses two specific directives, `!DIR$ PARALLEL` and `!DIR$ NOPARALLEL`.

Auto-parallelization Directives Format and Syntax

The format of Intel Fortran auto-parallelization compiler directive is:

```
<prefix> <directive>
```

where the brackets above mean:

- `<xxx>:` the prefix and directive are required

For fixed form source input, the prefix is `!DIR$` or `CDIR$`

For free form source input, the prefix is `!DIR$` only.

The prefix is followed by the directive name; for example:

```
!DIR$ PARALLEL
```

Since auto-parallelization directives begin with an exclamation point, the directives take the form of comments if you omit the `-parallel` option.

Examples

The `!DIR$ PARALLEL` directive instructs the compiler to ignore dependencies which it assumes may exist and which would prevent correct parallelization in the immediately following loop. However, if dependencies are proven, they are not ignored.

The `!DIR$ NOPARALLEL` directive disables auto-parallelization for the immediately following loop.

```
program main
parameter (n=100)
integer x(n),a(n)

!DIR$ NOPARALLEL
do i=1,n
x(i) = i
enddo

!DIR$ PARALLEL
do i=1,n
```

```
do i=1,n
a( x(i) ) = i
enddo
end
```

Auto-parallelization Environment Variables

Option	Description	Default
OMP_NUM_THREADS	Controls the number of threads used.	Number of processors currently installed in the system while generating the executable
OMP_SCHEDULE	Specifies the type of runtime scheduling.	static

Auto-parallelization Threashold Control and Diagnostics

Threshold Control

The -par_threshold{n} option sets a threshold for the auto-parallelization of loops based on the probability of profitable execution of the loop in parallel. The value of n can be from 0 to 100. The default value is 75. This option is used for loops whose computation work volume cannot be determined at compile-time. The threshold is usually relevant when the loop trip count is unknown at compile-time.

The -par_threshold{n} option has the following versions and functionality:

- Default: -par_threshold is not specified in the command line, which is the same as when -par_threshold0 is specified. The loops get auto-parallelized regardless of computation work volume, that is, parallelize always.
- -par_threshold100 - loops get auto-parallelized only if profitable parallel execution is almost certain.
- The intermediate 1 to 99 values represent the percentage probability for profitable speed-up. For example, n=50 would mean: parallelize only if there is a 50% probability of the code speeding up if executed in parallel.
- The default value of n is n=75 (or -par_threshold75). When -par_threshold is used on the command line without a number, the default value passed is 75.

The compiler applies a heuristic that tries to balance the overhead of creating multiple threads versus the amount of work available to be shared amongst the threads.

Diagnostics

The -par_report{0|1|2|3} option controls the auto-parallelizer's diagnostic levels 0, 1, 2, or 3 as follows:

-par_report0 = no diagnostic information is displayed.

-par_report1 = indicates loops successfully auto-parallelized (default). Issues a "LOOP AUTO-PARALLELIZED" message for parallel loops.

-par_report2 = indicates successfully auto-parallelized loops as well as unsuccessful loops.

-par_report3 = same as 2 plus additional information about any proven or assumed dependences inhibiting auto-parallelization (reasons for not parallelizing).

Example of Parallelization Diagnostics Report

Example below shows an output generated by -par_report3 as a result from the command:

prompt>**ifl -c /Qparallel /Qpar_report3 *myprog.f90***

where the program myprog.f90 is as follows:

```
 program myprog
      integer a(10000), q
C Assumed side effects
      do i=1,10000
         a(i) = foo(i)
      enddo
C Actual dependence
      do i=1,10000
         a(i) = a(i-1) + i
      enddo
      end
```

Example of -par_report Output

```
program myprog
 procedure: myprog
 serial loop: line 5: not a parallel candidate
          due to statement at line 6
 serial loop: line 9
      flow data dependence from line 10 to line
  10, due to "a"
12 Lines Compiled
```

Troubleshooting Tips

- Use `-par_threshold0` to see if the compiler assumed there was not enough computational work
- Use `-par_report3` to view diagnostics
- Use `!DIR$ PARALLEL` directive to eliminate assumed data dependencies
- Use `-ipo` to eliminate assumed side-effects done to function calls.

Debugging Multithreaded Programs

The debugging of multithreaded program discussed in this section applies to both the OpenMP Fortran API and the Intel Fortran parallel compiler directives. When a program uses parallel decomposition directives, you must take into consideration that the bug might be caused either by an incorrect program statement or it might be caused by an incorrect parallel decomposition directive. In either case, the program to be debugged can be executed by multiple threads simultaneously.

To debug the multithreaded programs, you can use:

- Intel Debugger for IA-32 and Intel Debugger for Itanium-based applications (idb)
- Intel Fortran Compiler debugging options and methods; in particular, Compiling Source Lines with Debugging Statements.
- Intel parallelization extension routines for low-level debugging.
- VTune(TM) Performance Analyzer to define the problematic areas.

Other best known debugging methods and tips include:

- Correct the program in single-threaded, uni-processor environment
- Statically analyze locks
- Use trace statement (such as `print` statement)
- Think in parallel, make very few assumptions
- Step through your code
- Make sense of threads and callstack information
- Identify the primary thread
- Know what thread you are debugging

- Single stepping in one thread does not mean single stepping in others

- Watch out for context switch

Debugger Limitations for Multithread Programs

Debuggers such as Intel Debugger for IA-32 and Intel Debugger for Itanium-based applications support the debugging of programs that are executed by multiple threads. However, the currently available versions of such debuggers do not directly support the debugging of parallel decomposition directives, and therefore, there are limitations on the debugging features.

Some of the new features used in OpenMP are not yet fully supported by the debuggers, so it is important to understand how these features work to know how to debug them. The two problem areas are:

- Multiple entry points

- Shared variables

You can use routine names (for example, `padd`) and entry names (for example, `_PADD`, `___PADD_6__par_loop0`). FORTRAN Compiler, by default, first mangles lower/mixed case routine names to upper case. For example, `pAdD()` becomes `PADD()`, and this becomes entry name by adding one underscore. The secondary entry name mangling happens after that. That's why "`__par_loop`" part of the entry name stays as lower case. Debugger for some reason didn't take the upper case routine name "`PADD`" to set the breakpoint. Instead, it accepted the lower case routine name "`padd`".

Debugging Parallel Regions

The compiler implements a parallel region by enabling the code in the region and putting it into a separate, compiler-created entry point. Although this is different from outlining – the technique employed by other compilers, that is, creating a subroutine, – the same debugging technique can be applied.

Constructing an Entry-point Name

The compiler-generated parallel region entry point name is constructed with a concatenation of the following strings:

- "__" character

- entry point name for the original routine (for example, `_parallel`)

- "_" character

- line number of the parallel region

- `__par_region` for OpenMP parallel regions (`!$OMP PARALLEL`)

  `__par_loop` for OpenMP parallel loops (`!$OMP PARALLEL DO`),

  `__par_section` for OpenMP parallel sections (`!$OMP PARALLEL SECTIONS`)

- sequence number of the parallel region (for each source file, sequence number starts from zero.)

Debugging Code with Parallel Region

Example 1 illustrates the debugging of the code with parallel region. Example 1 is produced by this command:

`ifc -openmp -g -O0 -S` ***`file.f90`***

Let us consider the code of subroutine `parallel`in Example 1.

Subroutine `PARALLEL()` source listing

```
1    subroutine parallel
2    integer id,OMP_GET_THREAD_NUM
3 !$OMP PARALLEL PRIVATE(id)
4    id = OMP_GET_THREAD_NUM()
5 !$OMP END PARALLEL
6    end
```

The parallel region is at line 3. The compiler created two entry points: `parallel_` and `___parallel_3__par_region0`. The first entry point corresponds to the subroutine `parallel()`, while the second entry point corresponds to the OpenMP parallel region at line 3.

Example 1 Debuging Code with Parallel Region

Machine Code Listing of the Subroutine `parallel()`

```
    .globl parallel_
parallel_:
..B1.1:                         # Preds ..B1.0
..LN1:
pushl     %ebp                                          #1.0
movl      %esp, %ebp                                    #1.0
subl      $44, %esp                                     #1.0
pushl     %edi                                          #1.0
movl      $.2.1_2_kmpc_loc_struct_pack.0, (%esp)  #1.0
call      __kmpc_global_thread_num                      #1.0
                        # LOE eax
..B1.21:                # Preds ..B1.1
addl      $4, %esp                                      #1.0
movl      %eax, -44(%ebp)                               #1.0
                        # LOE
```

```
                    # LOE
..B1.2:                 # Preds ..B1.21
 movl      -44(%ebp), %eax                               #1.0
 movl      %eax, -24(%ebp)                               #1.0
..LN2:
 pushl     %edi                                          #3.0
 movl      $.2.1_2_kmpc_loc_struct_pack.1, (%esp)        #3.0
 call      __kmpc_ok_to_fork                             #3.0
                    # LOE eax
..B1.22:                # Preds ..B1.2
 addl      $4, %esp                                      #3.0
 movl      %eax, -40(%ebp)                               #3.0
                    # LOE
..B1.3:                 # Preds ..B1.22
 movl      -40(%ebp), %eax                               #3.0
 testl     %eax, %eax                                    #3.0
 jne       ..B1.7      # Prob 50%                        #3.0
                    # LOE
..B1.4:                 # Preds ..B1.3
 addl      $-8, %esp                                     #3.0
 movl      $.2.1_2_kmpc_loc_struct_pack.1, (%esp)        #3.0
 movl      -24(%ebp), %eax                               #3.0
 movl      %eax, 4(%esp)                                 #3.0
 call      __kmpc_serialized_parallel                    #3.0
                    # LOE
..B1.23:                    # Preds ..B1.4
 addl      $8, %esp                                      #3.0
                    # LOE
..B1.5:                 # Preds ..B1.23
 addl      $-8, %esp                                     #3.0
 lea       -24(%ebp), %eax                               #3.0
 movl      %eax, (%esp)                                  #3.0
 movl      $___kmpv_zeroparallel__0, 4(%esp)             #3.0
 call      _parallel__3__par_region0                     #3.0
                     # LOE
..B1.24:                # Preds ..B1.5
 addl      $8, %esp                                      #3.0
                    # LOE
..B1.6:                 # Preds ..B1.24
 addl      $-8, %esp                                     #3.0
 movl      $.2.1_2_kmpc_loc_struct_pack.1, (%esp)        #3.0
 movl      -24(%ebp), %eax                               #3.0
 movl      %eax, 4(%esp)                                 #3.0
 call      __kmpc_end_serialized_parallel                #3.0
                     # LOE
..B1.25:                # Preds ..B1.6
 addl      $8, %esp                                      #3.0
 jmp       ..B1.8      # Prob 100%                       #3.0
                     # LOE
..B1.7:                 # Preds ..B1.3
 addl      $-12, %esp                                    #3.0
 movl      $.2.1_2_kmpc_loc_struct_pack.1, (%esp)        #3.0
 movl      $0, 4(%esp)                                   #3.0
 movl      $.parallel_3_par_region0_8(%esp)              #3.0
```

```
 movl      $_parallel__6__par_region0, 8(%esp)     #3.0
 call      __kmpc_fork_call                        #3.0
                     # LOE
..B1.26:              # Preds ..B1.7
 addl      $12, %esp                               #3.0
                     # LOE
..B1.8:               # Preds ..B1.26 ..B1.25
..LN3:
 pushl     %edi                                    #6.0
 movl      $.2.1_2_kmpc_loc_struct_pack.2, (%esp) #6.0
 call      __kmpc_ok_to_fork                       #6.0
                     # LOE eax
..B1.27:              # Preds ..B1.8
 addl      $4, %esp                                #6.0
 movl      %eax, -36(%ebp)                         #6.0
                     # LOE
..B1.9:               # Preds ..B1.27
 movl      -36(%ebp), %eax                         #6.0
 testl     %eax, %eax                              #6.0
 jne       ..B1.13       # Prob 50%                #6.0
                     # LOE
..B1.10:              # Preds ..B1.9
 addl      $-8, %esp                               #6.0
 movl      $.2.1_2_kmpc_loc_struct_pack.2, (%esp) #6.0
 movl      -24(%ebp), %eax                         #6.0
 movl      %eax, 4(%esp)                           #6.0
 call      __kmpc_serialized_parallel              #6.0
                     # LOE
..B1.28:              # Preds ..B1.10
 addl      $8, %esp                                #6.0
                     # LOE
..B1.11:              # Preds ..B1.28
 addl      $-8, %esp                               #6.0
 lea       -24(%ebp), %eax                         #6.0
 movl      %eax, (%esp)                            #6.0
 movl      $___kmpv_zeroparallel__1, 4(%esp)       #6.0
 call      _parallel__6__par_region1               #6.0
                     # LOE
..B1.29:              # Preds ..B1.11
 addl      $8, %esp                                #6.0
                     # LOE
..B1.12:              # Preds ..B1.29
 addl      $-8, %esp                               #6.0
 movl      $.2.1_2_kmpc_loc_struct_pack.2, (%esp) #6.0
 movl      -24(%ebp), %eax                         #6.0
 movl      %eax, 4(%esp)                           #6.0
 call      __kmpc_end_serialized_parallel          #6.0
                     # LOE
..B1.30:              # Preds ..B1.12
 addl      $8, %esp                                #6.0
 jmp       ..B1.14       # Prob 100%               #6.0
                     # LOE
..B1.13:              # Preds ..B1.9
 addl      $-12, %esp                              #6.0
```

```
 addl      $ 12, %esp                                    #6.0
 movl      $.2.1_2_kmpc_loc_struct_pack.2, (%esp) #6.0
 movl      $0, 4(%esp)                                   #6.0
 movl      $_parallel__6__par_region1, 8(%esp)    #6.0
 call      __kmpc_fork_call                              #6.0
                       # LOE
..B1.31:                # Preds ..B1.13
 addl      $12, %esp                                     #6.0
                       # LOE
..B1.14:                # Preds ..B1.31 ..B1.30
..LN4:
 leave                                                   #9.0
 ret                                                     #9.0
                       # LOE
.type parallel_,@function
.size parallel_,.-parallel_
.globl _parallel__3__par_region0
_parallel__3__par_region0:
# parameter 1: 8 + %ebp
# parameter 2: 12 + %ebp
..B1.15:                # Preds ..B1.0
 pushl     %ebp                                          #9.0
 movl      %esp, %ebp                                    #9.0
 subl      $44, %esp                                     #9.0
..LN5:
 call      omp_get_thread_num_                           #4.0
                       # LOE eax
..B1.32:                # Preds ..B1.15
 movl      %eax, -32(%ebp)                               #4.0
                       # LOE
..B1.16:                # Preds ..B1.32
 movl      -32(%ebp), %eax                               #4.0
 movl      %eax, -20(%ebp)                               #4.0
..LN6:
 leave                                                   #9.0
 ret                                                     #9.0
                       # LOE
.type _parallel__3__par_region0,@function
.size
_parallel__3__par_region0,._parallel__3__par_region0
.globl _parallel__6__par_region1
_parallel__6__par_region1:
# parameter 1: 8 + %ebp
# parameter 2: 12 + %ebp
..B1.17:                # Preds ..B1.0
 pushl     %ebp                                          #9.0
 movl      %esp, %ebp                                    #9.0
 subl      $44, %esp                                     #9.0
..LN7:
 call      omp_get_thread_num_                           #7.0
                       # LOE eax
..B1.33:                # Preds ..B1.17
 movl      %eax, -28(%ebp)                               #7.0
                       # LOE
```

```
                                # LOE
..B1.18:                        # Preds ..B1.33
 movl      -28(%ebp), %eax                               #7.0
 movl      %eax, -16(%ebp)                               #7.0
..LN8:
 leave                                                   #9.0
 ret                                                     #9.0
 .align    4,0x90
# mark_end;
```

Debugging the program at this level is just like debugging a program that uses POSIX threads directly. Breakpoints can be set in the threaded code just like any other routine. With GNU debugger, breakpoints can be set to source-level routine names (such as parallel). Breakpoints can also be set to entry point names (such as parallel_ and _parallel__3__par_region0). Note that Intel Fortran Compiler for Linux converted the upper case Fortran subroutine name to the lower case one.

Debugging Multiple Threads

When in a debugger, you can switch from one thread to another. Each thread has its own program counter so each thread can be in a different place in the code. Example 2 shows a Fortran subroutine `PADD()`. A breakpoint can be set at the entry point of OpenMP parallel region.

Source listing of the Subroutine `PADD()`

```
12.       SUBROUTINE PADD(A, B, C, N)
13.       INTEGER N
14.       INTEGER A(N), B(N), C(N)
15.       INTEGER I, ID, OMP_GET_THREAD_NUM
16. !$OMP PARALLEL DO SHARED (A, B, C, N) PRIVATE(ID)
17.       DO I = 1, N
18.         ID = OMP_GET_THREAD_NUM()
19.         C(I) = A(I) + B(I) + ID
20.       ENDDO
21. !$OMP END PARALLEL DO
22.       END
```

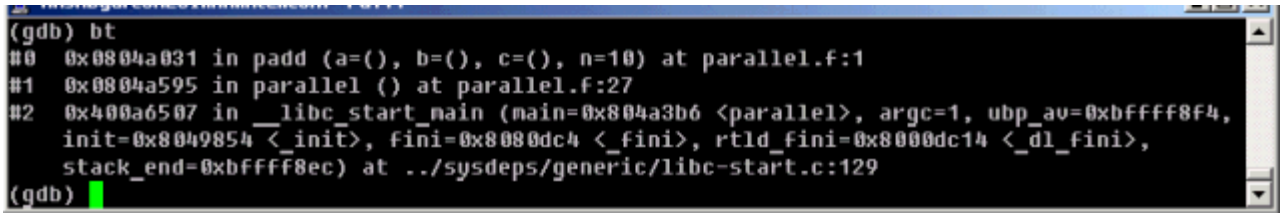
The Call Stack Dumps

The first call stack below is obtained by breaking at the entry to subroutine `PADD` using GNU debugger. At this point, the program has not executed any OpenMP regions, and therefore has only one thread. The call stack shows a system runtime `__libc_start_main` function calling the Fortran main program `parallel()`, and `parallel()` calls subroutine `padd()`. When the program is executed by more than one thread, you can switch from one thread to another. The second and the third call stacks are obtained by breaking at the entry to the parallel region. The call stack of master contains the complete call sequence. At the top of the call stack is `_padd__6__par_loop0()`. Invocation of a threaded entry point involves a layer of Intel OpenMP library function calls (that is, functions with `__kmp` prefix). The call stack of the worker thread contains a partial call sequence that begins with a layer of Intel OpenMP library function calls.

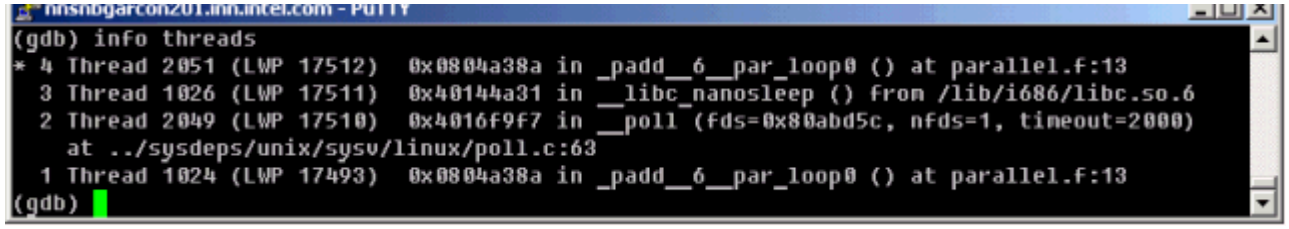
call sequence that begins with a layer of Intel OpenMP library function calls.

ERRATA: GNU debugger sometimes fails to properly unwind the call stack of the immediate caller of Intel OpenMP library function `__kmpc_fork_call()`.

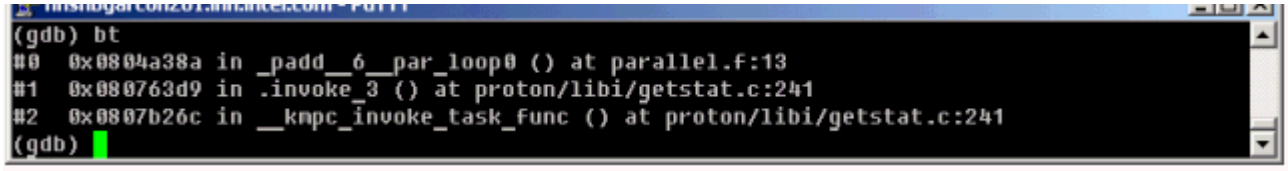
Call Stack Dump of Master Thread upon Entry to Subroutine PADD



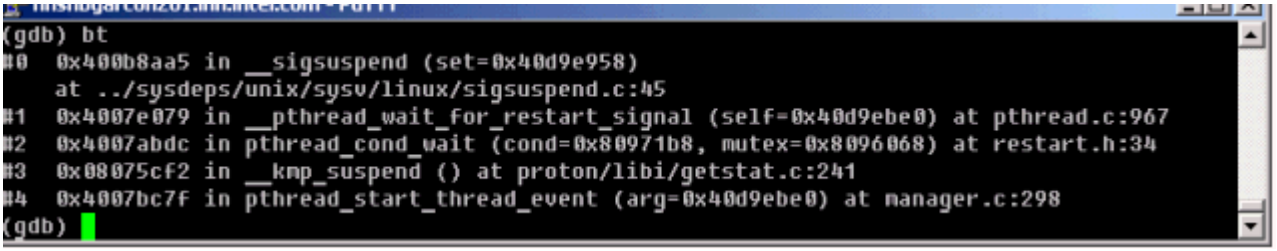
Switching from One Thread to Another



Call Stack Dump of Master Thread upon Entry to Parallel Region



Call Stack Dump of Worker Thread upon Entry to Parallel Region



Example 2 Debugging Code Using Multiple Threads with Shared Variables

Subroutine PADD() Machine Code Listing

`.globl padd_`
`padd_:`
`# parameter 1: 8 + %ebp`

```
# parameter 2: 12 + %ebp
# parameter 3: 16 + %ebp
# parameter 4(n): 20 + %ebp
..B1.1:                         # Preds ..B1.0
..LN1:
pushl     %ebp                                      #1.0
movl      %esp, %ebp                                #1.0
subl      $208, %esp                                #1.0
movl      %ebx, -4(%ebp)                            #1.0
pushl     %edi                                      #1.0
movl      $.2.1_2_kmpc_loc_struct_pack.0, (%esp) #1.0
call      __kmpc_global_thread_num                  #1.0
                          # LOE eax
..B1.34:                        # Preds ..B1.1
addl      $4, %esp                                  #1.0
movl      %eax, -28(%ebp)                           #1.0
                          # LOE
..B1.2:                         # Preds ..B1.34
movl      -28(%ebp), %eax                           #1.0
movl      %eax, -208(%ebp)                          #1.0
movl      $4, %eax                                  #1.0
movl      %eax, -184(%ebp)                          #1.0
movl      %eax, -188(%ebp)                          #1.0
movl      20(%ebp), %eax                            #1.0
movl      (%eax), %eax                              #1.0
movl      %eax, -24(%ebp)                           #1.0
testl     %eax, %eax                                #1.0
jg        ..B1.5        # Prob 50%                  #1.0
                        # LOE
..B1.3:                     # Preds ..B1.2
   movl      $0, -24(%ebp)                              #1.0
                        # LOE
..B1.5:                     # Preds ..B1.2 ..B1.3
movl      -24(%ebp), %eax                           #1.0
movl      %eax, -164(%ebp)                          #1.0
movl      $1, %eax                                  #1.0
movl      %eax, -176(%ebp)                          #1.0
movl      %eax, -168(%ebp)                          #1.0
movl      20(%ebp), %edx                            #1.0
movl      (%edx), %edx                              #1.0
movl      %edx, -172(%ebp)                          #1.0
movl      -164(%ebp), %edx                          #1.0
movl      %edx, -192(%ebp)                          #1.0
movl      8(%ebp), %edx                             #1.0
movl      %edx, -196(%ebp)                          #1.0
movl      $4, -204(%ebp)                            #1.0
movl      -204(%ebp), %edx                          #1.0
negl      %edx                                      #1.0
addl      -196(%ebp), %edx                          #1.0
movl      %edx, -200(%ebp)                          #1.0
movl      %eax, -180(%ebp)                          #1.0
movl      -192(%ebp), %eax                          #1.0
testl     %eax, %eax                                #1.0
```

```
testl     %eax, %eax                                    #1.0
jg        ..B1.8        # Prob 50%                      #1.0
                        # LOE
..B1.6:                       # Preds ..B1.5
movl      -172(%ebp), %eax                              #1.0
testl     %eax, %eax                                    #1.0
jg        ..B1.8        # Prob 50%                      #1.0
                        # LOE
..B1.7:                       # Preds ..B1.6
movl      $0, -172(%ebp)                                #1.0
                        # LOE
..B1.8:                       # Preds ..B1.6 ..B1.7 ..B1.5
movl      $4, %eax                                      #1.0
movl      %eax, -140(%ebp)                              #1.0
movl      %eax, -144(%ebp)                              #1.0
movl      $1, %edx                                      #1.0
movl      %edx, -132(%ebp)                              #1.0
movl      %edx, -124(%ebp)                              #1.0
movl      20(%ebp), %ecx                                #1.0
movl      (%ecx), %ecx                                  #1.0
movl      %ecx, -128(%ebp)                              #1.0
movl      -164(%ebp), %ecx                              #1.0
movl      %ecx, -148(%ebp)                              #1.0
movl      12(%ebp), %ecx                                #1.0
movl      %ecx, -152(%ebp)                              #1.0
movl      %eax, -160(%ebp)                              #1.0
movl      -160(%ebp), %eax                              #1.0
negl      %eax                                          #1.0
addl      -152(%ebp), %eax                              #1.0
movl      %eax, -156(%ebp)                              #1.0
movl      %edx, -136(%ebp)                              #1.0
movl      -148(%ebp), %eax                              #1.0
testl     %eax, %eax                                    #1.0
jg        ..B1.11       # Prob 50%                      #1.0
                        # LOE
..B1.9:                       # Preds ..B1.8
movl      -128(%ebp), %eax                              #1.0
testl     %eax, %eax                                    #1.0
jg        ..B1.11       # Prob 50%                      #1.0
                        # LOE
..B1.10:                      # Preds ..B1.9
movl      $0, -128(%ebp)                                #1.0
                        # LOE
..B1.11:                      # Preds ..B1.9 ..B1.10 ..B1.8
movl      $4, %eax                                      #1.0
movl      %eax, -100(%ebp)                              #1.0
movl      %eax, -104(%ebp)                              #1.0
movl      $1, %edx                                      #1.0
movl      %edx, -92(%ebp)                               #1.0
movl      %edx, -84(%ebp)                               #1.0
movl      20(%ebp), %ecx                                #1.0
movl      (%ecx), %ecx                                  #1.0
movl      %ecx, -88(%ebp)                               #1.0
movl      -164(%ebp), %ecx                              #1.0
```

```
movl      -104(%ebp), %ecx                              #1.0
movl      %ecx, -108(%ebp)                              #1.0
movl      16(%ebp), %ecx                                #1.0
movl      %ecx, -112(%ebp)                              #1.0
movl      %eax, -120(%ebp)                              #1.0
movl      -120(%ebp), %eax                              #1.0
negl      %eax                                          #1.0
addl      -112(%ebp), %eax                              #1.0
movl      %eax, -116(%ebp)                              #1.0
movl      %edx, -96(%ebp)                               #1.0
movl      -108(%ebp), %eax                              #1.0
testl     %eax, %eax                                    #1.0
jg        ..B1.14       # Prob 50%                      #1.0
                        # LOE
..B1.12:                   # Preds ..B1.11
movl      -88(%ebp), %eax                               #1.0
testl     %eax, %eax                                    #1.0
jg        ..B1.14       # Prob 50%                      #1.0
                        # LOE
..B1.13:                   # Preds ..B1.12
movl      $0, -88(%ebp)                                 #1.0
                        # LOE
..B1.14:                   # Preds ..B1.12 ..B1.13 ..B1.11
..LN2:
pushl     %edi                                          #6.0
movl      $.2.1_2_kmpc_loc_struct_pack.1, (%esp)        #6.0
call      __kmpc_ok_to_fork                             #6.0
                      # LOE eax
..B1.35:                 # Preds ..B1.14
addl      $4, %esp                                      #6.0
movl      %eax, -20(%ebp)                               #6.0
                      # LOE
..B1.15:                 # Preds ..B1.35
movl      -20(%ebp), %eax                               #6.0
testl     %eax, %eax                                    #6.0
jne       ..B1.19       # Prob 50%                      #6.0
                        # LOE
..B1.16:                   # Preds ..B1.15
addl      $-8, %esp                                     #6.0
movl      $.2.1_2_kmpc_loc_struct_pack.1, (%esp)        #6.0
movl      -208(%ebp), %eax                              #6.0
movl      %eax, 4(%esp)                                 #6.0
call      __kmpc_serialized_parallel                    #6.0
                      # LOE
..B1.36:                 # Preds ..B1.16
addl      $8, %esp                                      #6.0
                      # LOE
..B1.17:                 # Preds ..B1.36
addl      $-24, %esp                                    #6.0
lea       -208(%ebp), %eax                              #6.0
movl      %eax, (%esp)                                  #6.0
movl      $___kmpv_zeropadd__0, 4(%esp)                 #6.0
movl      -196(%ebp), %eax                              #6.0
movl      %eax, 8(%esp)                                 #6.0
```

```
movl      %eax, 8(%esp)                                 #6.0
movl      -152(%ebp), %eax                              #6.0
movl      %eax, 12(%esp)                                #6.0
movl      -112(%ebp), %eax                              #6.0
movl      %eax, 16(%esp)                                #6.0
lea       20(%ebp), %eax                                #6.0
movl      %eax, 20(%esp)                                #6.0
call      _padd__6__par_loop0                           #6.0
                          # LOE
..B1.37:                     # Preds ..B1.17
addl      $24, %esp                                     #6.0
                          # LOE
..B1.18:                     # Preds ..B1.37
addl      $-8, %esp                                     #6.0
movl      $.2.1_2_kmpc_loc_struct_pack.1, (%esp)        #6.0
movl      -208(%ebp), %eax                              #6.0
movl      %eax, 4(%esp)                                 #6.0
call      __kmpc_end_serialized_parallel                #6.0
                          # LOE
..B1.38:                     # Preds ..B1.18
addl      $8, %esp                                      #6.0
jmp       ..B1.31         # Prob 100%                   #6.0
                          # LOE
..B1.19:                     # Preds ..B1.15
addl      $-28, %esp                                    #6.0
movl      $.2.1_2_kmpc_loc_struct_pack.1, (%esp)        #6.0
movl      $4, 4(%esp)                                   #6.0
movl      $_padd__6__par_loop0, 8(%esp)                 #6.0
movl      -196(%ebp), %eax                              #6.0
movl      %eax, 12(%esp)                                #6.0
movl      -152(%ebp), %eax                              #6.0
movl      %eax, 16(%esp)                                #6.0
movl      -112(%ebp), %eax                              #6.0
movl      %eax, 20(%esp)                                #6.0
lea       20(%ebp), %eax                                #6.0
movl      %eax, 24(%esp)                                #6.0
call      __kmpc_fork_call                              #6.0
                          # LOE
..B1.39:                     # Preds ..B1.19
addl      $28, %esp                                     #6.0
jmp       ..B1.31         # Prob 100%                   #6.0
                          # LOE
..B1.20:                     # Preds ..B1.30
movl      $1, %eax                                      #6.0
movl      %eax, -72(%ebp)                               #6.0
..LN3:
movl      -80(%ebp), %edx                               #10.0
..LN4:
movl      %edx, -68(%ebp)                               #6.0
..LN5:
movl      -80(%ebp), %edx                               #10.0
..LN6:
movl      %edx, -64(%ebp)                               #6.0
movl      $0, -60(%ebp)                                 #6.0
```

```
movl      $0, -60(%ebp)                                 #6.0
movl      %eax, -56(%ebp)                               #6.0
addl      $-36, %esp                                    #6.0
movl      $.2.1_2_kmpc_loc_struct_pack.1, (%esp)        #6.0
movl      -8(%ebp), %edx                                #6.0
movl      %edx, 4(%esp)                                 #6.0
movl      $34, 8(%esp)                                  #6.0
lea       -60(%ebp), %edx                               #6.0
movl      %edx, 12(%esp)                                #6.0
lea       -72(%ebp), %edx                               #6.0
movl      %edx, 16(%esp)                                #6.0
lea       -68(%ebp), %edx                               #6.0
movl      %edx, 20(%esp)                                #6.0
lea       -56(%ebp), %edx                               #6.0
movl      %edx, 24(%esp)                                #6.0
movl      %eax, 28(%esp)                                #6.0
movl      %eax, 32(%esp)                                #6.0
call      __kmpc_for_static_init_4                      #6.0
                          # LOE
..B1.40:                     # Preds ..B1.20
addl      $36, %esp                                     #6.0
                          # LOE
..B1.21:                     # Preds ..B1.40
movl      -72(%ebp), %eax                               #6.0
movl      -64(%ebp), %edx                               #6.0
cmpl      %edx, %eax                                    #6.0
jg        ..B1.26         # Prob 50%                    #6.0
                          # LOE
..B1.22:                     # Preds ..B1.21
movl      -68(%ebp), %eax                               #6.0
movl      -64(%ebp), %edx                               #6.0
cmpl      %edx, %eax                                    #6.0
jg        ..B1.24         # Prob 50%                    #6.0
                          # LOE
..B1.23:                     # Preds ..B1.22
movl      -68(%ebp), %eax                               #6.0
movl      %eax, -16(%ebp)                               #6.0
jmp       ..B1.25         # Prob 100%                   #6.0
                          # LOE
..B1.24:                     # Preds ..B1.22
movl      -64(%ebp), %eax                               #6.0
movl      %eax, -16(%ebp)                               #6.0
                          # LOE
..B1.25:                     # Preds ..B1.24 ..B1.23
movl      -16(%ebp), %eax                               #6.0
movl      %eax, -68(%ebp)                               #6.0
movl      -72(%ebp), %eax                               #6.0
movl      %eax, -76(%ebp)                               #6.0
jmp       ..B1.27         # Prob 100%                   #6.0
                          # LOE
..B1.26:                     # Preds ..B1.28 ..B1.21
addl      $-8, %esp                                     #6.0
movl      $.2.1_2_kmpc_loc_struct_pack.1, (%esp)        #6.0
movl      -8(%ebp), %eax                                #6.0
```

```
movl      8(%ebp), %eax                                 #6.0
movl      %eax, 4(%esp)                                 #6.0
call      __kmpc_for_static_fini                        #6.0
                          # LOE
..B1.41:                     # Preds ..B1.26
addl      $8, %esp                                      #6.0
jmp       ..B1.31         # Prob 100%                   #6.0
                          # LOE
..B1.27:                     # Preds ..B1.28 ..B1.25
..LN7:
call      omp_get_thread_num_                           #8.0
                           # LOE eax
..B1.42:                      # Preds ..B1.27
movl      %eax, -12(%ebp)                               #8.0
                           # LOE
..B1.28:                      # Preds ..B1.42
movl      -12(%ebp), %eax                               #8.0
movl      %eax, -52(%ebp)                               #8.0
..LN8:
movl      -76(%ebp), %eax                               #9.0
..LN9:
movl      16(%ebp), %edx                                #6.0
..LN10:
movl      -76(%ebp), %ecx                               #9.0
..LN11:
movl      20(%ebp), %ebx                                #6.0
..LN12:
movl      -4(%ebx,%ecx,4), %ecx                         #9.0
addl      -4(%edx,%eax,4), %ecx                         #9.0
addl      -52(%ebp), %ecx                               #9.0
movl      -76(%ebp), %eax                               #9.0
..LN13:
movl      24(%ebp), %edx                                #6.0
..LN14:
movl      %ecx, -4(%edx,%eax,4)                         #9.0
..LN15:
incl      -76(%ebp)                                     #10.0
movl      -76(%ebp), %eax                               #10.0
movl      -68(%ebp), %edx                               #10.0
cmpl      %edx, %eax                                    #10.0
jle       ..B1.27       # Prob 50%                      #10.0
jmp       ..B1.26       # Prob 100%                     #10.0
                          # LOE
.type padd_,@function
.size padd_,.-padd_
.globl _padd__6__par_loop0
_padd__6__par_loop0:
# parameter 1: 8 + %ebp
# parameter 2: 12 + %ebp
# parameter 3: 16 + %ebp
# parameter 4: 20 + %ebp
# parameter 5: 24 + %ebp
# parameter 6: 28 + %ebp
..B1.30:                      # Preds ..B1.0
```

```
..B1.30:                        # Preds ..B1.0
..LN16:
pushl     %ebp                                          #13.0
movl      %esp, %ebp                                    #13.0
subl      $208, %esp                                    #13.0
movl      %ebx, -4(%ebp)                                #13.0
..LN17:
movl      8(%ebp), %eax                                 #6.0
movl      (%eax), %eax                                  #6.0
movl      %eax, -8(%ebp)                                #6.0
movl      28(%ebp), %eax                                #6.0
..LN18:
movl      (%eax), %eax                                  #7.0
movl      (%eax), %eax                                  #7.0
movl      %eax, -80(%ebp)                               #7.0
movl      $1, -76(%ebp)                                 #7.0
movl      -80(%ebp), %eax                               #7.0
testl     %eax, %eax                                    #7.0
jg        ..B1.20       # Prob 50%                      #7.0
                        # LOE
..B1.31:                   #
Preds ..B1.41 ..B1.39 ..B1.38 ..B1.30
..LN19:
movl      -4(%ebp), %ebx                                #13.0
leave                                                   #13.0
ret                                                     #13.0
.align    4,0x90
# mark_end;
```

Debugging Shared Variables

When a variable appears in a `PRIVATE`, `FIRSTPRIVATE`, `LASTPRIVATE`, or `REDUCTION` clause on some block, the variable is made private to the parallel region by redeclaring it in the block. `SHARED` data, however, is not declared in the threaded code. Instead, it gets its declaration at the routine level. At the machine code level, these shared variables become incoming subroutine call arguments to the threaded entry points (such as `___PADD_6__par_loop0`).

In Example 2, the entry point `___PADD_6_par_loop0` has six incoming parameters. The corresponding OpenMP parallel region has four shared variables. First two parameters (parameters 1 and 2) are reserved for the compiler's use, and each of the remaining four parameters corresponds to one shared variable. These four parameters exactly match the last four parameters to `__kmpc_fork_call()` in the machine code of `PADD`.

> **Note**
> The `FIRSTPRIVATE`, `LASTPRIVATE`, and `REDUCTION` variables also require shared variables to get the values into or out of the parallel region.

Due to the lack of support in debuggers, the correspondence between the shared variables (in their original names) and their contents cannot be seen in the debugger at the threaded entry point level. However, you can still move to the call stack of one of the subroutines and

examine the contents of the variables at that level. This technique can be used to examine the contents of shared variables. In Example 2, contents of the shared variables `A`, `B`, `C`, and `N` can be examined if you move to the call stack of `PARALLEL()`.

Vectorization

The vectorizer is a component of the Intel® Fortran Compiler that automatically uses SIMD instructions in the MMX(TM), SSE, and SSE2 instruction sets. The vectorizer detects operations in the program that can be done in parallel, and then converts the sequential operations like one SIMD instruction that processes 2, 4, 8 or up to 16 elements in parallel, depending on the data type.

This section provides options description, guidelines, and examples for Intel® Fortran Compiler vectorization implemented by IA-32 compiler only. For additional information, see Publications on Compiler Optimizations.

The following list summarizes this section contents.

- Descriptions of compiler options to control vectorization
- Vectorization Key Programming Guidelines
- Discussion and general guidelines on vectorization levels:

  —automatic vectorization

  —vectorization with user intervention
- Examples demonstrating typical vectorization issues and resolutions

The Intel compiler supports a variety of directives that can help the compiler to generate effective vector instructions. See compiler directives supporting vectorization.

Vectorizer Options

Vectorization is an IA-32-specific feature and can be summarized by the command line options described in the following tables. Vectorization depends upon the compiler's ability to disambiguate memory references. Certain options may enable the compiler to do better vectorization. These options can enable other optimizations in addition to vectorization. When a `-x{M|K|W}` or `-ax{M|K|W}` is used and `-O2` (which is ON by default) is also in effect, the vectorizer is enabled. The `-Qx{M|K|W}` or `-Qax{M|K|W}` options enable vectorizer with `-O1` and `-O3` options also.

`-x{M|K|W}`	Generate specialized code to run exclusively on the processors supporting the extensions indicated by `{M|K|W}`. See

	the extensions indicated by {M|K|W}. See Exclusive Specialized Code with -x{i|M|K|W} for details. **Note** -xi is not a vectorizer option.
-ax{M|K|W}	Generates, in a single binary, code specialized to the extensions specified by {M|K|W} and also generic IA-32 code. The generic code is usually slower. See Specialized Code with -ax{i|M|K|W} for details. **Note** -axi is not a vectorizer option.
-vec_report{0|1|2|3|4|5} Default: -vec_report1	Controls the diagnostic messages from the vectorizer, see subsection that follows the table.

Vectorization Reports

The -vec_report{0|1|2|3|4|5} options directs the compiler to generate the vectorization reports with different level of information as follows:

-vec_report0: no diagnostic information is displayed

-vec_report1: display diagnostics indicating loops successfully vectorized (default)

-vec_report2: same as -vec_report1, plus diagnostics indicating loops not successfully vectorized

-vec_report3: same as -vec_report2, plus additional information about any proven or assumed dependences
-vec_report4: indicate non-vectorized loops
-vec_report5: indicate non-vectorized loops and the reason why they were not vectorized.

Usage with Other Options

The vectorization reports are generated in the final compilation phase when executable is generated. Therefore if you use the -c option and a -vec_report{n} option in the command line, no report will be generated.

If you use -c, -ipo and -x{M|K|W} or -ax{M|K|W} and -vec_report{n}, the compiler

issues a warning and no report is generated.

To produce a report when using the above mentioned options, you need to add the `-ipo_obj` option. The combination of `-c` and `-ipo_obj` produces a single file compilation, and hence does generate object code, and eventually a report is generated.

The following commands generate vectorization report:

```
prompt>ifc -x{M|K|W} -vec_report3 file.f

prompt>ifc -x{M|K|W} -ipo -ipo_obj -vec_report3 file.f

prompt>ifc -c -x{M|K|W} -ipo -ipo_obj -vec_report3 file.f
```

Loop Parallelization and Vectorization

Combining the `-parallel` and `-x{M|K|W}` options instructs the compiler to attempt both automatic loop parallelization and automatic loop vectorization in the same compilation. In most cases, the compiler will consider outermost loops for parallelization and innermost loops for vectorization. If deemed profitable, however, the compiler may even apply loop parallelization and vectorization to the same loop. See Guidelines for Effective Auto-parallelization Usage and Vectorization Key Programming Guidelines.

Note that in some rare cases successful loop parallelization (either automatically or by means of OpenMP* directives) may affect the messages reported by the compiler for a non-vectorizable loop in a non-intuitive way.

Vectorization Key Programming Guidelines

The goal of vectorizing compilers is to exploit single-instruction multiple data (SIMD) processing automatically. Users can help however by supplying the compiler with additional information; for example, directives. Review these guidelines and restrictions, see code examples in further topics, and check them against your code to eliminate ambiguities that prevent the compiler from achieving optimal vectorization.

Guidelines

You will often need to make some changes to your loops.

For loop bodies -

Use:

- Straight-line code (a single basic block)
- Vector data only; that is, arrays and invariant expressions on the right hand side of

- Vector data only, that is, arrays and invariant expressions on the right hand side of assignments. Array references can appear on the left hand side of assignments.
- Only assignment statements

Avoid:

- Function calls
- Unvectorizable operations (other than mathematical)
- Mixing vectorizable types in the same loop
- Data-dependent loop exit conditions
- Loop unrolling (compiler does it)
- Decomposing one loop with several statements in the body into several single-statement loops.

Restrictions

Vectorization depends on the two major factors:

- **Hardware.** The compiler is limited by restrictions imposed by the underlying hardware. In the case of Streaming SIMD Extensions, the vector memory operations are limited to `stride-1` accesses with a preference to 16-byte-aligned memory references. This means that if the compiler abstractly recognizes a loop as vectorizable, it still might not vectorize it for a distinct target architecture.
- **Style.** The style in which you write source code can inhibit optimization. For example, a common problem with global pointers is that they often prevent the compiler from being able to prove that two memory references refer to distinct locations. Consequently, this prevents certain reordering transformations.

Many stylistic issues that prevent automatic vectorization by compilers are found in loop structures. The ambiguity arises from the complexity of the keywords, operators, data references, and memory operations within the loop bodies.

However, by understanding these limitations and by knowing how to interpret diagnostic messages, you can modify your program to overcome the known limitations and enable effective vectorization. The following sections summarize the capabilities and restrictions of the vectorizer with respect to loop structures.

Data Dependence

Data dependence relations represent the required ordering constraints on the operations in serial loops. Because vectorization rearranges the order in which operations are executed, any auto-vectorizer must have at its disposal some form of data dependence analysis.

any auto-vectorizer must have at its disposal some form of data dependence analysis.

An example where data dependencies prohibit vectorization is shown below. In this example, the value of each element of an array is dependent on the value of its neighbor that was computed in the previous iteration.

Data-dependent Loop

```
REAL DATA(0:N)
INTEGER I
DO I=1, N-1
DATA(I) =DATA(I-1)*0.25+DATA(I)*0.5+DATA(I+1)*0.25
END DO
```

The loop in the above example is not vectorizable because the `WRITE` to the current element `DATA(I)` is dependent on the use of the preceding element `DATA(I-1)`, which has already been written to and changed in the previous iteration. To see this, look at the access patterns of the array for the first two iterations as shown below.

Data Dependence Vectorization Patterns

```
I=1: READ DATA (0)
READ DATA (1)
READ DATA (2)
WRITE DATA (1)
I=2: READ DATA(1)
READ DATA (2)
READ DATA (3)
WRITE DATA (2)
```

In the normal sequential version of this loop, the value of `DATA(1)` read from during the second iteration was written to in the first iteration. For vectorization, it must be possible to do the iterations in parallel, without changing the semantics of the original loop.

Data Dependence Analysis

Data dependence analysis involves finding the conditions under which two memory accesses may overlap. Given two references in a program, the conditions are defined by:

- whether the referenced variables may be aliases for the same (or overlapping) regions in memory, and, for array references
- the relationship between the subscripts

For IA-32, data dependence analyzer for array references is organized as a series of tests, which progressively increase in power as well as in time and space costs. First, a number of simple tests are performed in a dimension-by-dimension manner, since independence in any dimension will exclude any dependence relationship. Multidimensional arrays references that may cross their declared dimension boundaries can be converted to their

linearized form before the tests are applied. Some of the simple tests that can be used are the fast greatest common divisor (GCD) test and the extended bounds test. The GCD test proves independence if the GCD of the coefficients of loop indices cannot evenly divide the constant term. The extended bounds test checks for potential overlap of the extreme values in subscript expressions. If all simple tests fail to prove independence, we eventually resort to a powerful hierarchical dependence solver that uses Fourier-Motzkin elimination to solve the data dependence problem in all dimensions. For more details of data dependence theory and data dependence analysis, refer to the Publications on Compiler Optimizations.

Loop Constructs

Loops can be formed with the usual `DO-ENDDO` and `DO WHILE`, or by using a `GOTO` and a label. However, the loops must have a single entry and a single exit to be vectorized. Following are the examples of correct and incorrect usages of loop constructs.

Correct Usage

```
SUBROUTINE FOO (A, B, C)
DIMENSION A(100),B(100), C
(100)
INTEGER I
I = 1
DO WHILE (I .LE. 100)
A(I) = B(I) * C(I)
IF (A(I) .LT. 0.0) A(I) =
0.0
I = I + 1
ENDDO
RETURN
END
```

Incorrect Usage

```
SUBROUTINE FOO (A, B, C)
DIMENSION A(100),B(100), C
(100)
INTEGER I
I = 1
DO WHILE (I .LE. 100)
A(I) = B(I) * C(I)
C The next statement allows
early
C exit from the loop and
prevents
C vectorization of the loop.
IF (A(I) .LT. 0.0) GOTO 10
```

```
I = I + 1
ENDDO
10 CONTINUE
RETURN
END
```

Loop Exit Conditions

Loop exit conditions determine the number of iterations that a loop executes. For example, fixed indexes for loops determine the iterations. The loop iterations must be countable; that is, the number of iterations must be expressed as one of the following:

- a constant
- a loop invariant term
- a linear function of outermost loop indices

Loops whose exit depends on computation are not countable. Examples below show countable and non-countable loop constructs.

Correct Usage for Countable Loop, Example 1

```
SUBROUTINE FOO (A, B, C, N, LB)
DIMENSION A(N),B(N),C(N)
INTEGER N, LB, I, COUNT
! Number of iterations is "N - LB +
1"
COUNT = N
DO WHILE (COUNT .GE. LB)
A(I) = B(I) * C(I)
COUNT = COUNT - 1
I = I + 1
ENDDO  ! LB is not defined within
loop
RETURN
END
```

Correct Usage for Countable Loop, Example 2

```
! Number of iterations is (N-M+2) /2
SUBROUTINE FOO (A, B, C, M, N, LB)
DIMENSION A(N),B(N),C(N)
INTEGER I, L, M, N
I = 1;
DO L = M,N,2
A(I) = B(I) * C(I)
I = I + 1
ENDDO
RETURN
END
```

Incorrect Usage for Non-countable Loop

```
! Number of iterations is dependent
on A(I)
SUBROUTINE FOO (A, B, C)
DIMENSION A(100),B(100),C(100)
INTEGER I
I = 1
DO WHILE (A(I) .GT. 0.0)
A(I) = B(I) * C(I)
I = I + 1
ENDDO
RETURN
END
```

Types of Loop Vectorized

For integer loops, the 64-bit MMX(TM) technology and 128-bit Streaming SIMD Extensions (SSE) provide SIMD instructions for most arithmetic and logical operators on 32-bit, 16-bit, and 8-bit integer data types. Vectorization may proceed if the final precision of integer wrap-around arithmetic will be preserved. A 32-bit shift-right operator, for instance, is not vectorized in 16-bit mode if the final stored value is a 16-bit integer. Because the MMX(TM) and SSE instruction sets are not fully orthogonal (shifts on byte operands, for instance, are not supported), not all integer operations can actually be vectorized.

For loops that operate on 32-bit single-precision and 64-bit double-precision floating-point numbers, SSE provides SIMD instructions for the arithmetic operators '+', '-', '*', and '/'. In addition, SSE provides SIMD instructions for the binary `MIN` and `MAX` and unary `SQRT` operators. SIMD versions of several other mathematical operators (like the trigonometric functions `SIN`, `COS`, `TAN`) are supported in software in a vector mathematical runtime library that is provided with the Intel® Fortran Compiler, of which the compiler takes advantage.

Stripmining and Cleanup

The compiler automatically strip-mines your loop and generates a cleanup loop.

Stripmining and Cleanup Loops

Before Vectorization

```
i = 1
do while (i<=n)
a(i) = b(i) + c(i) ! Original loop code
i = i + 1
end do
```

After Vectorization

```
!The vectorizer generates the following
two loops
i = 1
do while (i < (n - mod(n,4)))
! Vector strip mined loop
```

```
! vector strip-mined loop.
a(i:i+3) = b(i:i+3) + c(i:i+3)
i = i + 4
end do
do while (i <= n)
a(i) = b(i) + c(i)     !Scalar clean-up
loop
i = i + 1
end do
```

Statements in the Loop Body

The vectorizable operations are different for floating point and integer data.

Floating-point Array Operations

The statements within the loop body may be `REAL` operations (typically on arrays). Arithmetic operations supported are addition, subtraction, multiplication, division, negation, square root, `MAX`, `MIN`, and mathematical functions such as `SIN` and `COS`. Note that conversion to/from some types of floats is not valid. Operation on `DOUBLE PRECISION` types is not valid, unless optimizing for a
Pentium® 4 and Xeon(TM) processors system, using the `-xW` or `-axW` compiler option.

Integer Array Operations

The statements within the loop body may be arithmetic or logical operations (again, typically for arrays). Arithmetic operations are limited to such operations as addition, subtraction, `ABS`, `MIN`, and `MAX`. Logical operations include bitwise `AND`, `OR` and `XOR` operators. You can mix data types only if the conversion can be done without a loss of precision. Some example operators where you can mix data types are multiplication, shift, or unary operators.

Other Operations

No statements other than the preceding floating-point and integer operations are permitted. The loop body cannot contain any function calls other than the ones described above.

Vectorization Examples

This section contains simple examples of some common issues in vector programming.

Argument Aliasing: A Vector Copy

The loop in the example of a vector copy operation does not vectorize because the compiler cannot prove that `DEST(A(I))` and `DEST(B(I))` are distinct.

Unvectorizable Copy Due to Unproven Distinction

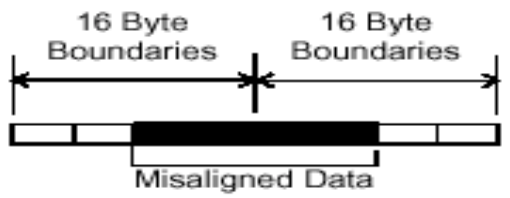
```
SUBROUTINE VEC_COPY
(DEST,A,B,LEN)
DIMENSION DEST(*)
INTEGER A(*), B(*)
INTEGER LEN, I
DO I=1,LEN
DEST(A(I)) = DEST(B(I))
END DO
RETURN
END
```

Data Alignment

A 16-byte or greater data structure or array should be aligned so that the beginning of each structure or array element is aligned in a way that its base address is a multiple of 16.

The Misaligned Data Crossing 16-Byte Boundary figure shows the effect of a data cache unit (DCU) split due to misaligned data. The code loads the misaligned data across a 16-byte boundary, which results in an additional memory access causing a six- to twelve-cycle stall. You can avoid the stalls if you know that the data is aligned and you specify to assume alignment

Misaligned Data Crossing 16-Byte Boundary



After vectorization, the loop is executed as shown in figure below.

Vector and Scalar Clean-up Iterations

Both the vector iterations `A(1:4) = B(1:4)`; and `A(5:8) = B(5:8)`; can be implemented with aligned moves if both the elements `A(1)` and `B(1)` are 16-byte aligned.

Caution

If you specify the vectorizer with incorrect alignment options, the compiler will generate code with unexpected behavior. Specifically, using aligned moves on unaligned data, will result in an illegal instruction exception!

Alignment Strategy

The compiler has at its disposal several alignment strategies in case the alignment of data structures is not known at compile-time. A simple example is shown below (several other strategies are supported as well). If in the loop shown below the alignment of A is unknown, the compiler will generate a prelude loop that iterates until the array reference, that occurs the most, hits an aligned address. This makes the alignment properties of A known, and the vector loop is optimized accordingly. In this case, the vectorizer applies dynamic loop peeling, a specific Intel® Fortran feature.

Data Alignment Example

Original loop:

```
SUBROUTINE DOIT(A)
REAL A(100)       ! alignment of argument A
is unknown
DO I = 1, 100
A(I) = A(I) + 1.0
ENDDO
END SUBROUTINE
```

Aligning Data

```
! The vectorizer will apply dynamic loop
peeling as follows:
SUBROUTINE DOIT(A)
REAL A(100)
! let P be (A%16)where A is address of A(1)
IF (P .NE. 0) THEN
P = (16 - P) /4    ! determine runtime
peeling factor
DO I = 1, P
A(I) = A(I) + 1.0
ENDDO
ENDIF
! Now this loop starts at a 16-byte boundary,

! and will be vectorized accordingly
DO I = P + 1, 100
A(I) = A(I) + 1.0
ENDDO
END SUBROUTINE
```

Loop Interchange and Subscripts: Matrix Multiply

Matrix multiplication is commonly written as shown in the following example.

```
DO I=1, N
DO J=1, N
DO K=1, N
C(I,J) = C(I,J) + A(I,K)*B(K,J)
```

```
END DO
END DO
END DO
```

The use of `B(K,J)`, is not a `stride-1` reference and therefore will not normally be vectorizable. If the loops are interchanged, however, all the references will become `stride-1` as in the Matrix Multiplication with Stride-1 example that follows.

Note

Interchanging is not always possible because of dependencies, which can lead to different results.

Matrix Multiplication with Stride-1

```
DO J=1,N
DO K=1,N
DO I=1,N
C(I,J) = C(I,J) + A(I,K)*B
(K,J)
ENDDO
ENDDO
ENDDO
```

For additional information, see Publications on Compiler Optimizations.

Optimization Support Features

This section describes the Intel® Fortran features such as directives, intrinsics, runtime library routines and various utilities which enhance your application performance in support of compiler optimizations. These features are Intel Fortran language extensions that enable you optimize your source code directly. This section includes examples of optimizations supported by Intel extended directives and intrinsics or library routines that enhance and/or help analyze performance.

For complete detail of the Intel® Fortran Compiler directives and examples of their use, see Appendix A in the *Intel® Fortran Programmer's Reference*. For intrinsic procedures, see Chapter 1, "Intrinsic Procedures," in the *Intel® Fortran Libraries Reference*.

A special topic describes options that enable you to generate optimization reports for major compiler phases and major optimizations. The optimization report capability is used for Itanium®-based applications only.

Compiler Directives

This section discusses the Intel® Fortran language extended directives that enhance optimizations of application code, such as software pipelining, loop unrolling, prefetching and vectorization. For complete list, descriptions and code examples of the Intel® Fortran Compiler directives, see Appendix A in the *Intel® Fortran Programmer's Reference*.

Pipelining for Itanium®-based Applications

The `SWP | NOSWP` directives indicate preference for a loop to get software-pipelined or not. The `SWP` directive does not help data dependence, but overrides heuristics based on profile counts or lop-sided control flow.

The syntax for this directive is:

`CDIR$ SWP` or `!DIR$ SWP`

`CDIR$ NOSWP` or `!DIR$ NOSWP`

The software pipelining optimization triggered by the `SWP` directive applies instruction scheduling to certain innermost loops, allowing instructions within a loop to be split into different stages, allowing increased instruction level parallelism. This can reduce the impact of long-latency operations, resulting in faster loop execution. Loops chosen for software pipelining are always innermost loops that do not contain procedure calls that are not inlined. Because the optimizer no longer considers fully unrolled loops as innermost loops, fully unrolling loops can allow an additional loop to become the innermost loop (see `-unroll[n]`). You can request and view the optimization report to see whether software pipelining was applied (see Optimizer Report Generation).

SWP

```
CDIR$ SWP
do i = 1, m
if (a(i) .eq. 0)  then
b(i) = a(i) + 1
else
b(i) = a(i)/c(i)
endif
enddo
```

LOOP COUNT (N) Directive

The `LOOP COUNT (n)` directive indicates the loop count is likely to be `n`.

The syntax for this directive is:

`CDIR$ LOOP COUNT(n)` or `!DIR$ LOOP COUNT(n)`

where n is an integer constant.

The value of loop count affects heuristics used in software pipelining, vectorization and loop-transformations.

LOOP COUNT (N)

```
CDIR$ LOOP COUNT (10000)
do i =1,m
b(i) = a(i) +1 ! This is likely to enable
        ! the loop to get software-
        ! pipelined
enddo
```

Loop Distribution Directive

The `DISTRIBUTE POINT` directive indicates to compiler a preference of performing loop distribution.

The syntax for this directive is:

`CDIR$ DISTRIBUTE POINT` or `!DIR$ DISTRIBUTE POINT`

Loop distribution may cause large loops be distributed into smaller ones. This may enable more loops to get software-pipelined. If the directive is placed inside a loop, the distribution is performed after the directive and any loop-carried dependency is ignored. If the directive is placed before a loop, the compiler will determine where to distribute and data dependency is observed. Currently only one distribute directive is supported if it is placed inside the loop.

DISTRIBUTE POINT

```
CDIR$ DISTRIBUTE POINT
do i =1, m
b(i) = a(i) +1
....
c(i) = a(i) + b(i) ! Compiler will decide where
          ! to distribute.
          ! Data dependency is observed
....
d(i) = c(i) + 1
enddo

do i =1, m
b(i) = a(i) +1
....
CDIR$  DISTRIBUTE POINT
call  sub(a, n)    ! Distribution will start
here,
          ! ignoring all loop-carried
          ! dependency
c(i) = a(i) + b(i)
....
d(i) = c(i) + 1
enddo
```

Loop Unrolling Support

The UNROLL directive tells the compiler how many times to unroll a counted loop.

The syntax for this directive is:

CDIR$ UNROLL or !DIR$ UNROLL

CDIR$ UNROLL [n] or !DIR$ UNROLL [n]

CDIR$ NOUNROLL or !DIR$ NOUNROLL

where n is an integer constant. The range of n is 0 through 255.

The UNROLL directive must precede the do statement for each do loop it affects.

If n is specified, the optimizer unrolls the loop n times. If n is omitted or if it is outside the allowed range, the optimizer assigns the number of times to unroll the loop.

The UNROLL directive overrides any setting of loop unrolling from the command line.

Currently, the directive can be applied only for the innermost loop nest. If applied to the outer loop nests, it is ignored. The compiler generates correct code by comparing n and the loop count.

UNROLL
CDIR$ UNROLL(4) do i = 1, m b(i) = a(i) + 1 d(i) = c(i) + 1 enddo

Prefetching Support

The PREFETCH and NOPREFTCH directives assert that the data prefetches be generated or not generated for some memory references. This affects the heuristics used in the compiler.

The syntax for this directive is:

CDIR$ PREFETCH or !DIR$ PREFETCH

CDIR$ NOPRFETCH or !DIR$ NOPREFETCH

CDIR$ PREFETCH a,b or !DIR$ PREFETCH a,b

If loop includes expression a(j), placing PREFETCH a in front of the loop, instructs the compiler to insert prefetches for a(j + d) within the loop. d is determined by the compiler. This directive is supported when option -O3 is on.

PREFETCH
CDIR$ NOPREFETCH c CDIR$ PREFETCH a do i = 1, m b(i) = a(c(i)) + 1 enddo

Vectorization Support (IA-32)

The directives discussed in this topic support vectorization and used for IA-32 applications only.

IVDEP Directive

The compiler supports IVDEP directive which instructs the compiler to ignore assumed vector dependences. Use this directive when you know that the assumed loop dependences are safe to ignore.

For example, if the expression j >= 0 is always true in the code fragment bellow, the IVDEP directive can communicate this information to the compiler. This directive informs the compiler that the conservatively assumed loop-carried flow dependences for values j < 0 can be safely ignored:

```
!DIR$ IVDEP
do i = 1, 100
a(i) = a(i+j)
enddo
```

Note

The proven dependeces that prevent vectorization are not ignored, only assumed dependeces are ignored.

The syntax for the directive is:

```
CDIR$IVDEP
!DIR$IVDEP
```

The usage of the directive differs depending on the loop form, see examples below.

Loop 1

`Do i`
`= a(*) + 1`
`a(*) =`
`enddo`

Loop 2

`Do i`
`a(*) =`
`= a(*) + 1`
`enddo`

For loops of the form 1, use old values of `a`, and assume that there is no loop-carried flow dependencies from `DEF` to `USE`.

For loops of the form 2, use new values of `a`, and assume that there is no loop-carried anti-dependencies from `USE` to `DEF`.

In both cases, it is valid to distribute the loop, and there is no loop-carried output dependency.

Example 1

`CDIR$IVDEP`
`do  j=1,n`
`a(j) = a(j+m) + 1`
`enddo`

Example 2

`CDIR$IVDEP`
`do  j=1,n`
`a(j) = b(j) +1`
`b(j) = a(j+m) + 1`
`enddo`

Example 1 ignores the possible backward dependencies and enables the loop to get software pipelined.

Example 2 shows possible forward and backward dependencies involving array a in this loop and creating a dependency cycle. With IVDEP, the backward dependencies are ignored.

IVDEP has options: IVDEP:LOOP and IVDEP:BACK. The IVDEP:LOOP option implies no loop-carried dependencies. The IVDEP:BACK option implies no backward dependencies.

The IVDEP directive is also used for Itanium®-based applications.

For more details on the IVDEP directive, see Appendix A in the *Intel® Fortran Programmer's Reference*.

Overriding Vectorizer's Efficiency Heuristics

In addition to IVDEP directive, there are three directives that can be used to override the efficiency heuristics of the vectorizer:

```
!DIR$VECTOR ALWAYS
!DIR$NOVECTOR
!DIR$VECTOR ALIGNED
!DIR$VECTOR UNALIGNED
```

The VECTOR ALWAYS directive overrides the efficiency heuristics of the vectorizer, but it only works if the loop can actually be vectorized, that is: use IVDEP to ignore assumed dependences.

The VECTOR ALWAYS and NOVECTOR Directives

The VECTOR ALWAYS directive can be used to override the default behavior of the compiler in the following situation. Vectorization of non-unit stride references usually does not exhibit any speedup, so the compiler defaults to not vectorizing loops that have a large number of non-unit stride references (compared to the number of unit stride references). The following loop has two references with stride 2. Vectorization would be disabled by default, but the directive overrides this behavior.

Vector Aligned
!DIR$ VECTOR ALWAYS do i = 1, 100, 2 a(i) = b(i) enddo

If, on the other hand, avoiding vectorization of a loop is desirable (if vectorization results in a performance regression rather than improvement), the NOVECTOR directive can be used in the source text to disable vectorization of a loop. For instance, the Intel® Compiler vectorizes the following example loop by default. If this behavior is not appropriate, the NOVECTOR directive can be used, as shown below.

NOVECTOR

`!DIR$ NOVECTOR`
`do i = 1, 100`
`a(i) = b(i) + c(i)`
`enddo`

The `VECTOR ALIGNED` and `UNALIGNED` Directives

Like `VECTOR ALWAYS`, these directives also override the efficiency heuristics. The difference is that the qualifiers `UNALIGNED` and `ALIGNED` instruct the compiler to use, respectively, unaligned and aligned data movement instructions for all array references. This disables all the advanced alignment optimizations of the compiler, such as determining alignment properties from the program context or using dynamic loop peeling to make references aligned.

Note

The directives `VECTOR` [`ALWAYS`, `UNALIGNED`, `ALIGNED`] should be used with care. Overriding the efficiency heuristics of the compiler should only be done if the programmer is absolutely sure the vectorization will improve performance. Furthermore, instructing the compiler to implement all array references with aligned data movement instructions will cause a runtime exception in case some of the access patterns are actually unaligned.

Compiler Intrinsics

Intel® Fortran supports all standard Fortran intrinsic procedures and in addition, provides Intel-specific intrinsic procedures to extend the functionality of the language. Intel Fortran intrinsic procedures are provided in the library `libintrins.lib`. See Chapter 1, "Intrinsic Procedures," in the *Intel® Fortran Libraries Reference*.

This topic provides examples of the Intel-extended intrinsics that are helpful in developing efficient applications.

Cache Size Intrinsic (Itanium® Compiler)

Intrinsic `cashesize(n)` is used only with Intel® Itanium® Compiler. `cashesize(n)` returns the size in kilobytes of the cache at level `n`; 1 represents the first level cache. Zero is returned for a nonexistent cache level.

This intrinsic can be used in many scenarios where application programmer would like to tailor their algorithms for target processor's cache hierarchy. For example, an application may query the cache size and use it to select block sizes in algorithms that operate on matrices.

```
subroutine foo (level)
integer level
if (cachesize(level) >
threshold)
call big_bar()
else
call small_bar()
end if
end subroutine
```

Timing Your Application

One of the performance indicators is your application timing. Use the `time` command to provide information about program performance. The following considerations apply to timing your application:

- Run program timings when other users are not active. Your timing results can be affected by one or more CPU-intensive processes also running while doing your timings.
- Try to run the program under the same conditions each time to provide the most accurate results, especially when comparing execution times of a previous version of the same program. Use the same CPU system (model, amount of memory, version of the operating system, and so on) if possible.
- If you do need to change systems, you should measure the time using the same version of the program on both systems, so you know each system's effect on your timings.
- For programs that run for less than a few seconds, run several timings to ensure that the results are not misleading. Overhead functions like loading shared libraries might influence short timings considerably.

Using the form of the `time` command that specifies the name of the executable program provides the following:

- The elapsed, real, or "wall clock" time, which will be greater than the total charged actual CPU time.
- Charged actual CPU time, shown for both system and user execution. The total actual CPU time is the sum of the actual user CPU time and actual system CPU time.

Example

In the following example timings, the sample program being timed displays the following line:

```
Average of all the numbers is:    4368488960.000000
```

Using the Bourne shell, the following program timing reports that the program uses 1.19 seconds of total actual CPU time (0.61 seconds in actual CPU time for user program use and 0.58 seconds of actual CPU time for system use) and 2.46 seconds of elapsed time:

```
$ time a.out

Average of all the numbers is:
  4368488960.000000

real    0m2.46s

user    0m0.61s

sys     0m0.58s
```

Using the C shell, the following program timing reports 1.19 seconds of total actual CPU time (0.61 seconds in actual CPU time for user program use and 0.58 seconds of actual CPU time for system use), about 4 seconds (0:04) of elapsed time, the use of 28% of available CPU time, and other information:

```
% time a.out

Average of all the numbers is:
  4368488960.000000

0.61u 0.58s 0:04 28% 78+424k 9+5io 0pf+0w
```

Using the bash shell, the following program timing reports that the program uses 1.19 seconds of total actual CPU time (0.61 seconds in actual CPU time for user program use and 0.58 seconds of actual CPU time for system use) and 2.46 seconds of elapsed time:

```
[user@system user]$ time ./a.out

Average of all the numbers is:
  4368488960.000000

elapsed  0m2.46s

user     0m0.61s

sys      0m0.58s
```

Timings that show a large amount of system time may indicate a lot of time spent doing I/O, which might be worth investigating.

If your program displays a lot of text, you can redirect the output from the program on the time command line. Redirecting output from the program will change the times reported because of reduced screen I/O.

For more information, see `time(1)`.

In addition to the `time` command, you might consider modifying the program to call routines within the program to measure execution time. For example, use the Intel Fortran intrinsic procedures, such as `SECNDS`, `DCLOCK`, `CPU_TIME`, `SYSTEM_CLOCK`, and `DATE_AND_TIME`. See "Intrinsic Procedures" in the *Intel® Fortran Libraries Reference*.

Optimizer Report Generation (Itanium® Compiler)

The Intel® Fortran Itanium® Compiler for Itanium®-based Applications provides options to generate and manage optimization reports.

- `-opt_report` generates optimizations report and places it in a file specified in `-opt_report_file`*`filename`*. If `-opt_report_file` is not specified, `-opt_report` directs the report to `stderr`. The default is OFF: no reports are generated.
- `-opt_report_file`*`filename`* generates optimizations report and directs it to a file specified in *`filename`*.
- `-opt_report_level{`*`min`*`|`*`med`*`|`*`max`*`}` specifies the detail level of the optimizations report. The *`min`* argument provides the minimal summary and the *`max`* the full report. The default is `-opt_report_level`*`min`*.
- `-opt_report_routine`*`routine_substring`* generates reports from all routines with names containing the *`substring`* as part of their name. If not specified, reports from all routines are generated. The default is to generate reports for all routines being compiled.

Specifying Optimizations to Generate Reports

The compiler can generate reports for an optimizer you specify in the *`phase`* argument of the `-opt_report_phase`*`phase`* option.

The option can be used multiple times on the same command line to generate reports for multiple optimizers.

Currently, the following optimizer reports are supported:

Optimizer Logical Name	Optimizer Full Name
`ipo`	Interprocedural Optimizer
`hlo`	High Level Optimizer

`ilo`	Intermediate Language Scalar Optimizer
`ecg`	Itanium Compiler Code Generator
`omp`	OpenMP*
`all`	All optimizers

When one of the above logical names for optimizers are specified all reports from that optimizer will be generated. For example, `-opt_report_phase`*`ipo`* and `-opt_report_phase`*`ecg`* generate reports from the interprocedural optimizer and the code generator.

Each of the optimizers can potentially have specific optimizations within them. Each of these optimizations are prefixed with one of the optimizer logical names. For example:

Optimizer_optimization	**Full Name**
`ipo_inline`	Interprocedural Optimizer, inline expansion of functions
`ipo_constant_propagation`	Interprocedural Optimizer, constant propagation
`ipo_function_reoder`	Interprocedural Optimizer, function reorder
`ilo_constant_propagation`	Intermediate Language Scalar Optimizer, constant propagation
`ilo_copy_propagation`	Intermediate Language Scalar Optimizer, copy propagation
`ecg_software_pipelining`	Itanium Compiler Code Generator, software pipelining

Command Syntax Example

The following command generates a report for the Itanium Compiler Code Generator (`ecg`):

prompt>**`efc -c -opt_report -opt_report_phase`** ***`ecg myfile.f`***

where:

- `-c` tells the compiler to stop at generating the object code, not linking
- `-opt_report` invokes the report generator
- `-opt_report_phase`*`ecg`* indicates the phase (`ecg`) for which to generate the report; the space between the option and the phase is optional.

The entire name for a particular optimization within an optimizer need not be specified in full, just a few characters is sufficient. All optimization reports that have a matching prefix

with the specified optimizer are generated. For example, if `-opt_report_phase` *`ilo_co`* is specified, a report from both the constant propagation and the copy propagation are generated.

The Availability of Report Generation

The `-opt_report_help` option lists the logical names of optimizers that are currently available for report generation.

Libraries

You can determine the libraries for your applications by controlling the linker or by using the options described in this section. See library options summary.

The LD_LIBRARY_PATH environment variable contains a colon-separated list of directories that the linker will search for library (.a) files. If you want the linker to search additional libraries, you can add their names to the command line, to a response file, or to the configuration (.cfg) file. In each case, the names of these libraries are passed to the linker before these libraries:

- the libraries provided with the Intel® Fortran Compiler (libCEPCF90.a, libIEPCF90.a, libintrins.a, libF90.a, and the math library: libimf.a for both IA-32 compiler and libm.a for Itanium® compiler; libm.a is the math library provided with the gcc*)

- the default libraries that the compiler command always specifies are:

  ```
  libimf.a *
  libm.a
  libirc.a *
  libcxa.a *
  libcprts.a *
  libunwind.a *
  libc.a
  ```

  The ones marked with an "*" are provided by Intel.

For more information on response and configuration files, see Response Files and Configuration Files.

The linker uses the LD_LIBRARY_PATH variable to search for libraries. If you are compiling with a linker option that forces static libraries, it will look for those at compile time. Otherwise, it will look for shared libraries at runtime.

To specify a library name on the command line, you must first add the library's path to the LD_LIBRARY_PATH environment variable. Then, to compile file.f and link it with the library libmine.a, for example, enter the following command:

IA-32 applications:

```
prompt>ifc file.f -lmine
```

Itanium®-based applications:

```
prompt>efc file.f -lmine
```

The example above implies that the library resides in your path.

The Order of Passing the Files to Linker

The compiler passes files to the linker in the following order:

1. Object files and libraries are passed to the linker in the order specified on the command line.

2. Object files and libraries in the `.cfg` file will be processed before those on the command line. This means that putting library names in the `.cfg` file does not make much sense because the libraries will be processed before most object files are seen.

3. The `libimf.a`, `libF90.a`, `libintrins.a`, and `libIEPCF90.a` libraries.

4. The `libm.a` library is linked in just before `libc.a`, then `libc.a` libraries.

See the list of libraries that are installed with the Intel® Fortran Compiler for IA-32 applications and for Itanium®-based applications.

Using the POSIX* and Portability Libraries

Use the `-posixlib` option with the compiler to invoke the POSIX* bindings library `libposf90.a`. For a complete list of these functions see Chapter 3, "POSIX Functions" in the *Intel® Fortran Libraries Reference Manual.*

Use the `-Vaxlib` option with the compiler to invoke the VAX* compatibility functions `libpepcf90.a`. This also brings in the Intel's compatibility functions for Sun* and Microsoft*. For a complete list of these functions see Chapter 2, "Portability Functions" in the *Intel® Fortran Libraries Reference Manual*.

Intel® Shared Libraries

The Intel® Fortran Compiler (both IA-32 and Itanium® compilers) links the libraries statically at link time and dynamically at the run time, the latter as dynamically shared objects (DSO).

By default, the libraries are linked as follows:

- Fortran, math and `libcprts.a` libraries are linked at link time, that is, statically.
- `libcxa.so` is linked dynamically to conform to C++ application binary interface (ABI).
- GNU and Linux* system libraries are linked dynamically.

Advantages of This Approach

This approach—

- Enables to maintain the same model for both IA-32 and Itanium compilers.
- Provides a model consistent with the Linux model where system libraries are dynamic and application libraries are static.
- The users have the option of using dynamic versions of our libraries to reduce the size of their binaries if desired.
- The users are licensed to distribute Intel-provided libraries.

The libraries `libcprts.a` and `libcxa.so` are C++ language support libraries used by Fortran when Fortran includes code written in C++.

Shared Library Options

The main options used with shared libraries are `-i_dynamic` and `-shared`.

The `-i_dynamic` compiler option directs the linker to use the shared object versions of the Intel-provided libraries dynamically. The comparison of the following commands illustrates the effects of this option.

1. *prompt*>**ifc *myprog.f***

This command produces the following results (default):

- Fortran, math, `libirc.a`, and `libcprts.a` libraries are linked statically (at link time).
- Dynamic version of `libcxa.so` is linked at run time.

The statically linked libraries increase the size of the application binary, but do not need to be installed on the systems where the application runs.

2. *prompt*>**ifc -i_dynamic *myprog.f***

This command links all of the above libraries dynamically. This has the advantage of reducing the size of the application binary, but it requires all the dynamic versions installed on the systems where the application runs.

The `-shared` option instructs the compiler to build a dynamically shared object (DSO) instead of an executable. For more details, refer to the `ld` man page documentation.

Math Libraries

The libimf.a is the math library provided by Intel and libm.a is the math library provided with gcc*. Both of these libraries are linked in by default on IA-32 and Itanium® compilers. Both libraries are linked in because there are math functions supported by the GNU math library that are not in the Intel math library. This linking arrangement allows for all functions GNU users have available to them to be available when using ifc (or efc), with Intel optimized versions available when supported. libimf.a is linked in before libm.a. If you link in libm.a first, it will change the versions of the math functions that are used.

It is recommended that you place libimf.a and libm.a in the first directory specified in the LD_LIBRARY_PATH variable. The libimf.a and libm.a libraries are always linked with Fortran programs.

For example, if you place a library in directory /perform/, set the LD_LIBRARY_PATH variable to specify a list of directories, containing all other libraries, separated by semicolons.

For IA-32 Compiler, libm.a contains both generic math routines and versions of the math routines optimized for special use with the Intel® Pentium® 4 and Xeon(TM) processors. For Itanium® Compiler, libm.a is optimized for the use with Itanium architecture.

IA-32 Compiler

For IA-32 Compiler, libm.lib contains both generic math routines and versions of the math routines optimized for special use with the Intel® Pentium® 4 and Xeon(TM) processors.

Itanium® Compiler

For Itanium Compiler, libm.lib is optimized for the use with Itanium® architecture. The Itanium compiler provides inlined version of the following math library primitives by using the following intrinsics: ALOG, DLOG, ALOG10, DLOG10, lEXP, DEXP, CEILING, and FLOOR. The compiler inlines these intrinsics and schedules the generated code with surrounding instructions. This can improve performance of typical floating-point applications.

Using Math Libraries with IA-32 Systems

Most of the routines in libm.a for IA-32 have been optimized for special use with the Intel® Pentium® 4 and Xeon(TM) processors. Generic versions are used when running on an IA-32 processor generation prior to Pentium 4 processor family.

To use your own version of the standard math functions without unresolved external errors, you must disable the automatic inline expansion by compiling your program with the -nolib_inline option, as described in Inline Expansion of Library Functions.

Caution

A change of the default precision control or rounding mode (for example, by using the `-pc32` flag or by user intervention) may affect the results returned by some of the mathematical functions.

Optimized Math Library Primitives

The optimized math libraries contain a package of functions, called primitives. The Intel Fortran Compiler calls these functions to implement numerous floating-point intrinsics and exponentiation. About half of the functions in the library from Intel are written in assembly language and optimized for program execution speed on an IA-32 architecture processor.

Note
The library primitives are not Fortran intrinsics. They are standard library calls used by the compiler to implement Intel Fortran language features.

Following is a list of math library primitives that have been optimized.

acos	cos	log10	sinh
asin	cosh	pow	sqrt
atan	exp	powf	tan
atan2	log	sin	tanh

The math library also provides the following non-optimized primitives.

acosh	copysign	fmod	gamma
asinh	erf	fmodf	remainder
atanh	fabs	hypot	rint
cbrt	fabsf	j0	y0
ceil	floor	j1	y1
ceilf	floorf	jn	y2

Programming with Math Library Primitives

Primitives adhere to standard calling conventions, thus you can call them with other high-level languages as well as with assembly language. For Intel Fortran Compiler programs, specify the appropriate Fortran intrinsic name for arguments of type `REAL` and `DOUBLE PRECISION`. The compiler calls the appropriate single- or double-precision primitive based on the type of the argument you specify.

To use these functions, you have to write an `INTERFACE` block that specifies the `ALIAS` name of the function. The routine names in the math library are lower case.

IEEE* Floating-point Exceptions

The compiler recognizes a set of floating-point exceptions required for compatibility with the IEEE numeric floating-point standard. The following floating-point exceptions are supported

during numeric processing:

Denormal	One of the floating-point operands has an absolute value that is too small to represent with full precision in the significand.
Zero Divide	The dividend is finite and the divisor is zero, but the correct answer has infinite magnitude.
Overflow	The resulting floating-point number is too large to represent.
Underflow	The resulting floating-point number (which is very close to zero) has an absolute value that is too small to represent even if a loss of precision is permitted in the significand (gradual underflow).
Inexact (Precision)	The resulting number is not represented exactly due to rounding or gradual underflow.
Invalid operation	Covers cases not covered by other exceptions. An invalid operation produces a quiet NaN (Not-a-Number).

Denormal

The denormal exception occurs if one or more of the operands is a denormal number. This exception is never regarded as an error.

Divide-by-Zero Exception

A divide-by-zero exception occurs for a floating-point division operation if the divisor is zero and the dividend is finite and non-zero. It also occurs for other operations in which the operands are finite and the correct answer is infinite.

When the divide by zero exception is masked, the result is +/-infinity. The following specific cases cause a zero-divide exception:

- `LOG(0.0)`
- `LOG10(0.0)`
- `0.0**x`, where `x` is a negative number

For the value of the flags, refer to the `ieee_flags ()` function in your library manual and *Pentium® Processor Family Developer's Manual*, Volumes 1, 2, and 3.

Overflow Exception

An overflow exception occurs if the rounded result of a floating-point operation contains an exponent larger than the numeric processing unit can represent. A calculation with an

infinite input number is not sufficient to cause an exception.

When the overflow exception is masked, the calculated result is +/-infinity or the +/-largest representable normal number depending on rounding mode. When the exception is not masked, a result with an accurate significand and a wrapped exponent is available to an exception handler.

Underflow Exception

The underflow exception occurs if the rounded result has an exponent that is too small to be represented using the floating-point format of the result.

If the underflow exception is masked, the result is represented by the smallest normal number, a denormal number, or zero. When the exception is not masked, a result with an accurate significand and a wrapped exponent is available to an exception handler

Inexact Exception

The inexact exception occurs if the rounded result of an operation is not equal to the unrounded result.

It is important that the inexact exception remain masked at all times because many of the numeric library procedures return with an undefined precision exception flag. If the precision exception is masked, no special action is performed. When this exception is not masked, the rounded result is available to an exception handler.

Invalid Operation Exception

An invalid operation indicates that an exceptional condition not covered by one of the other exceptions has occurred. An invalid operation can be caused by any of the following situations:

- One or more of the operands is a signaling NaN or is in an unsupported format.
- One of the following invalid operations has been requested:

  `(+--)0.0-(+--)0.0, (+--)0.0*(+--)?, or (+--)?-(+--)?.`

- The function `INT`, `NINT`, or `IRINT` is applied to an operand that is too large to fit into the requested `INTEGER*2` or `INTEGER*4` data types.
- A comparison of `.LT.`, `.LE.`, `.GT.`, or `.GE.` is applied to two operands that are unordered.

The invalid-operation exception can occur in any of the following functions:

- `SQRT(x)`, `LOG(x)`, or `LOG10(x)`, where `x` is less than zero.
- `ASIN(x)`, or `ACOS(x)` where `|x|>1`.

For any of the invalid-operation exceptions, the exception handler is invoked before the top of the stack changes, so the operands are available to the exception handler.

When invalid-operation exceptions are masked, the result of an invalid operation is a quiet NaN. Program execution proceeds normally using the quiet NaN result.

Floating-point Result	The appearance of a quiet NaN as an operand results in a quiet NaN. Execution continues without an error. If both operands are quiet NaNs, the quiet NaN with the larger significand is used as the result. Thus, each quiet NaN is propagated through later floating-point calculations until it is ultimately ignored or referenced by an operation that delivers non-floating-point results.
Formatted Output	On formatted output using a real edit descriptor, the field is filled with the "?" symbols to indicate the undefined (NaN) result. The `A`, `Z`, or `B` edit descriptor results in the ASCII, hexadecimal, or binary interpretation, respectively, of the internal representation of the NaN. No error is signaled for output of a NaN.
Logical Result	By definition, a NaN has no ordinal rank with respect to any other operand, even itself. Tests for equality (`.EQ.`) and inequality (`.NE.`) are the only Fortran relational operations for which results are defined for unordered operands. In these cases, program execution continues without error. Any other logical operation yields an undefined result when applied to NaNs, causing an invalid-operation error. The masked result is unpredictable.
Integer Result	Since no internal NaN representation exists for the `INTEGER` data type, an invalid-operation error is normally signaled. The masked result is the largest-magnitude negative integer for `INTEGER*4` or `INTEGER*2`. An `INTEGER*1` result is the value of an `INTEGER*2` intermediate result modulo 256.

Intel® Fortran Compiler provides a method to control the rounding mode, exception handling and other IEEE-related functions of the IA-32 processors using `IEEE_FLGS` and `IEEE_HANDLER` library routines from the portability library. For details, see Chapter 2 in the *Intel® Fortran Libraries Reference Manual*.

Compiler Diagnostics

This section describes the diagnostic messages that the Intel® Fortran Compiler produces. These messages include various diagnostic messages for remarks, warnings, or errors. The compiler always displays any error message, along with the erroneous source line, on the standard error device. The messages also include the runtime diagnostics run for IA-32 compiler only.

The options that provide checks and diagnostic information must be specified when the program is compiled, but they perform checks or produce information when the program is run. See diagnostic options summary.

Runtime Diagnostics

For IA-32 applications, the Intel® Fortran Compiler provides runtime diagnostic checks to aid debugging. The compiler provides a set of options that identify certain conditions commonly attributed to runtime failures.

You must specify the options when the program is compiled. However, they perform checks or produce information when the program is run. Postmortem reports provide additional diagnostics according to the detail you specify.

Runtime diagnostics are handled by IA-32 options only. The use of `-O0` option turns any of them off. See the runtime check options summary.

Optional Runtime Checks

Runtime checks on the use of pointers, allocatable arrays and assumed-shape arrays are made with the runtime checks specified by the Intel® Fortran Compiler command line runtime diagnostic options listed below. The use of any of these options disables optimization.

The optional runtime check options are as follows:

`-C`	Equivalent to: (`-CA`, `-CB`, `-CS`, `-CU`, `-CV`) **Note** The `-C` option and its equivalents are available for IA-32 systems only.
`-CA`	Should be used in conjunction with `-d{n}`. Generates runtime code, which checks pointers and allocatable array references for `nil`. **Note** The run-time checks on the use of pointers, allocatable arrays and assumed-shape arrays are made if

	compile-time option `-CA` is selected.
`-CB`	Should be used in conjunction with `-d{n}`. Generates runtime code to check that array subscript and substring references are within declared bounds.
`-CS`	Should be used in conjunction with `-d{n}`. Generates runtime code that checks for consistent shape of intrinsic procedure.
`-CU`	Should be used in conjunction with `-d{n}`. Generates runtime code that causes a runtime error if variables are used without being initialized.
`-CV`	Should be used in conjunction with `-d{n}`. On entry to a subprogram, tests the correspondence between the actual arguments passed and the dummy arguments expected. Both calling and called code must be compiled with `-CV` for the checks to be effective.

Pointers, `-CA`

The selection of the `-CA` compile-time option has the following effect on the runtime checking of pointers:

- The association status of a pointer is checked whenever it is referenced. Error 460 as described in Runtime Errors will be reported at runtime if the pointer is disassociated: that is, if the pointer is nullified, de-allocated, or it is a pointer assigned to a disassociated pointer.

- The compile-time option combination of `-CA` and `-CU` also generates code to test whether a pointer is in the initially undefined state, that is, if it has never been associated or disassociated or allocated. If a pointer is initially undefined, then Error 461 as described in Runtime Errors will be reported at runtime if an attempt is made to use it. No test is made for dangling pointers (that is, pointers referencing memory locations which are no longer valid).

- The association status of pointers is not tested when the Fortran standard does not require the pointer to be associated, that is, in the following circumstances:

  - in a pointer assignment

  - as an argument to the `associated` intrinsic

  - as an argument to the `present` intrinsic

  - in the `nullify` statement

  - as an actual argument associated with a formal argument which has the pointer attribute

Allocatable Arrays

The selection of the `-CA` compile-time option causes code to be generated to test the allocation status of an allocatable array whenever it is referenced, except when it is an argument to the `allocated` intrinsic function. Error 459 as described in Runtime Errors will be reported at runtime if an error is detected.

Assumed-Shape Arrays

The `-CA` option causes a validation check to be made, on entry to a procedure, on the definition status of an assumed-shape array. Error 462 as described in Runtime Errors will be reported at runtime if the array is disassociated or not allocated.

The compile-time option combination of `-CA` and `-CU` will additionally generate code to test whether, on entry to a procedure, the array is in the initially undefined state. If so, Error 463 as described in Runtime Errors.

Array Subscripts, Character Substrings, `-CB`

Specifying the compile-time option `-CB` causes a check at runtime that array subscript values, subscript values of elements selected from an array section, and character substring references are within bounds. Selection of the option causes code to be generated for each array or character substring reference in the program.

At runtime the code checks that the address computed for a referenced array element is within the address range delimited by the first element of the array and the last element of the array. Note that this check does not ensure that each subscript in a reference to an element of a multidimensional array or section is within bounds, only that the address of the element is within the address range of the array.

For assumed-size arrays, only the address of the first element of the array is used in the check; the address of the last element is unknown.

When `-CB` is selected, a check is also made that any character substring references are within the bounds of the character entity referenced.

Unassigned Variables, `-CU`

Specifying the compile-time option `-CU` causes unassigned variable checking to be enabled: that is, before an expression is evaluated at runtime, a check is normally made that any variables in the expression have previously been assigned values. If any has not, a runtime error results.

Some variables are not unassigned-checked, even when `-CU` has been selected:

- Variables of type `character`
- `byte`, `integer(1)` and `logical(1)` variables

- Variables of derived type, when the complete variable (not individual fields) is used in the expression
- Arguments passed to some elemental and transformational intrinsic procedures

Notes on Variables

- Variables that specify storage with `allocate`, except those of types noted in the previous section, will be unassigned-checked when `-CU` is selected.
- If the variables in a named `COMMON` block are to be unassigned-checked, `-CU` must be selected, and:

  - The `COMMON` block must be specified in one and only one `BLOCK DATA` program unit. Variables in the `COMMON` block that are not explicitly initialized will be subject to the unassigned check.

  - No variable of the `COMMON` block may be initialized outside the `BLOCK DATA` program unit.
- Variables in blank `COMMON` will be subject to the unassigned check if `-CU` is selected and the blank `COMMON` appears in the main program unit. In this case, although the Intel® Fortran Compiler permits blank `COMMON` to have different sizes in different program units, only the variables within the extent of blank `COMMON` indicated in the main program unit will be subject to the unassigned check.

Actual to Dummy Argument Correspondence, `-CV`

Specifying the compile-time option `-CV` causes checks to be carried out at runtime that actual arguments to subprograms correspond with the dummy arguments expected. Note the following:

- Both caller and called Fortran code must be compiled with `-CV` (or `-C`). No argument checking will be performed unless this condition is satisfied.
- The amount of checking performed depends upon whether the procedure call was made via an implicit interface or an explicit interface. Irrespective of the type of interface used, however, the following checks verify that:

  - the correct number of arguments are passed.

  - the type and type kinds of the actual and dummy arguments correspond.

  - subroutines have been called as subroutines and that functions have been declared with the correct type and type kind.

  - dummy arrays are associated with either an array or an element of an array and not a scalar variable or constant.

- the declared length of a dummy character argument is not greater than the declared length of associated actual argument.

- the declared length of a character scalar function result is the same length as that declared by the caller.

- the actual and dummy arguments of derived type correspond to the number and types of the derived type components.

- actual arguments were not passed using the intrinsic procedures %REF and %VAL.

- If an implicit interface call was made, then yet another check is made whether an interface block should have been used.

- If an explicit interface block was used, then further checks are made in addition to those described (in the second bullet) above, to validate the interface block. These checks verify that:

- the OPTIONAL attribute of each dummy argument has been correctly specified by the caller.

- the POINTER attribute of each dummy argument has been correctly specified by the caller.

- the declared length of a dummy pointer of type character is the same as the declared length of the associated actual pointer of type character.

- the rank of an assumed-shape array or dummy pointer matches the rank of the associated actual argument.

- the rank of an array-valued function or pointer-valued function has been correctly specified by the caller.

- the declared length of a character array-valued function or a character pointer-valued function is the same length as that declared by the caller.

Diagnostic Report, -d{n}

The command option -d{n} generates the additional information required for a list of the current values of variables to be output when certain runtime errors occur. Diagnostic reports are generated by the following:

- input/output errors
- an invalid reference to a pointer or an allocatable array (if -CA option selected)
- subscripts out of bounds (if -CB option selected)
- an invalid array argument to an intrinsic procedure (if -CS option selected)
- use of unassigned variables (if -CU option selected)
- argument mismatch (if -CV option selected)

- invalid assigned labels
- a call to the abort routine
- certain mathematical errors reported by intrinsic procedures
- hardware detected errors

The Level of Output

The level of output is progressively controlled by `n`, as follows:

n=0 (or n omitted)	Displays only the procedure name and the number of the line at which the failure occurred. This is the default value.
n=1	Reports scalar variables local to program active units.
n=2	Reports local and `COMMON` scalars.
n>2	Reports the first n elements of local and `COMMON` arrays and all scalars.

The appropriate error message will be output on `stderr`, and (if selected) a postmortem report will be produced.

Selecting a Postmortem Report

Each scalar or array will be displayed on a separate line in a form appropriate to the type of the variable. Thus, for example, variables of type integer will be output as integer values, and variables of type complex will be output as complex values.

The postmortem report will not include those program units which are currently active, but which have not been compiled with the `-d{n}` option. If no active program unit has been compiled with the `-d{n}` option then no postmortem report will be produced.

> **Note**
> Using the `-d{n}` option for postmortem reports disables optimization.

Invoking a Postmortem Report

A postmortem report may be invoked by any of the following:

- an error detected as a consequence of using the `-CA`, `-CB`, `-CS`, `-CU`, `-CV` or `-C` options
- a call on abort
- an allocation error
- an invalid assigned label

- an input-output error
- an error reported by a mathematical procedure
- a signal generated by a program error such as illegal instruction
- an error reported by an intrinsic procedure

Postmortem Report Conventions

The following conventions are used in postmortem output:

- A variable `var` declared in a module `mod` appears as `mod.var`.
- A module procedure `proc` in module `mod` appears as `mod$proc`.
- The fields of a variable `var` of derived data type are preceded by a line of the form `var%`.

Example

In this example, the command line

prompt`>ifc -CB -CU -d4` ***`sample.f`***

is used to compile the program that follows. When the program is executed, the postmortem report (follows the program) is output, since the subscript `m` to array `num` is out of bounds.

The Program

```
1 module arith
2 integer count
3 data count /0/
4
5 contains
6
7  subroutine add(k,p,m)
8   integer num(3),p
9
10   count = count+1
11   m = k+p
12   j = num(m)
13   return
14  end subroutine
15
16 end module arith
17
18 program dosums
```

```
19  use arith
20  type set
21    integer sum, product
22  end type set
23
24  type(set) ans
25
26  call add(9,6,ans%sum)
27
28 end program dosums
```

The Postmortem Report

```
Run-Time Error 406: Array bounds
exceeded
In Procedure: arith$add
Diagnostics Entered From Subroutine
arith$add Line 12
j       =   Not Assigned
k       =   9
m       =   15
num     =   Not Assigned, Not
Assigned, Not Assigned
p       =   6
Module arith
arith.count = 1
Entered From MAIN PROGRAM  Line  26
ans%
sum      =   15
product  =   Not Assigned
arith.count = 1
```

Compiler Information Messages

These messages are generated by the following Intel® Fortran Compiler options:

Disabling the sign-on message	
`-nologo`	Disables the display of the compiler version (or sign-on) message. When you sign-on, the compiler displays the following information: `ID`: the unique identification number for this compiler. `x.y.z`: the version of the compiler. `years`: the years for which the software is copyrighted.
Printing the list and brief description of the compiler driver options	

`-help`	You can print a list and brief description of the most useful compiler driver options by specifying the `-help` option to the compiler. To print this list, use this command: **IA-32 compiler:** *prompt*`>ifc -help` or *prompt*`>ifc -?` **Itanium® compiler:** *prompt*`>efc -help` or *prompt*`>efc -?`
Showing compiler version and driver tool commands	
`-V`	Displays compiler version information.
`-v`	Shows driver tool commands and executes tools.
`-dryrun`	Shows driver tool commands, but does not execute tools.

Diagnostic Messages

Diagnostic messages provide syntactic and semantic information about your source text. Syntactic information can include, for example, syntax errors and use of non-ANSI Fortran. Semantic information includes, for example, unreachable code.

Diagnostic messages can be any of the following: command-line diagnostics, warning messages, error messages, or catastrophic error messages.

Command-line Diagnostics

These messages report improper command-line options or arguments. If the command line contains an unrecognized option, the compiler passes the option to the linker. If the linker still does not recognize the option, the linker produces the diagnostic message.

Command-line error messages appear on the standard error device in the form:

`driver-name: `*`message`*

where

`driver-name`	The name of the compiler driver.
`message`	Describes the error.

Command-line warning messages appear as follows:

`driver-name``: warning: `*`message`*

Language Diagnostics

These messages describe diagnostics that are reported during the processing of the source

file. These diagnostics have the following format:

filename(linenum): type *nn: message*

filename	Indicates the name of the source file currently being processed. An extension to the filename indicates the type of the source file, as follows: `.f, f90, .for` indicate a Fortran file.
linenum	Indicates the source line where the compiler detects the condition.
type	Indicates the severity of the diagnostic message: warning, error, or Fatal error.
nn	The number assigned to the error (or warning) message.
message	Describes the diagnostic.

The following is an example of a warning message:

```
tantst.f(3): warning 328:"local variable": Local variable
"increment" never used.
```

The compiler can also display internal error messages on the standard error device. If your compilation produces any internal errors, contact your Intel representative. Internal error messages are in the form:

```
FATAL COMPILER ERROR: message
```

Warning Messages

These messages report valid but questionable use of the language being compiled. The compiler displays warnings by default. You can suppress warning messages by using the `-W0` option. Warnings do not stop translation or linking. Warnings do not interfere with any output files. Some representative warning messages are:

```
constant truncated - precision too great

non-blank characters beyond column 72 ignored

Hollerith size exceeds that required by the context
```

Suppressing or Enabling Warning Messages

The warning messages report possible errors and use of non-standard features in the source file.

The following options suppress or enable warning messages.

`-cerrs[-]`	Causes error and warning messages to be generated in a terse format: `"file", line no : error message` `-cerrs-` disables `-cerrs`.
`-w`	Suppresses all warning messages.
`-w90`, `-w95`	Suppresses warning messages about Fortran features which are deprecated or obsoleted in Fortran 95.
`-W{n}`	Suppresses or displays all warning messages generated by preprocessing and compilation. n=0: suppresses all warnings n=1: displays warning messages. `-W1` is the default.
`-WB`	On a bound check violation, issues a warning instead of an error. (This is to accommodate old FORTRAN code, in which array bounds of dummy arguments were frequently declared as 1.)

For example, the following command compiles `newprog.f` and displays compiler errors, but not warnings:

IA-32 compiler:

prompt>**ifc -W0 *newprog.f***

Itanium® compiler:

prompt>**efc -W0 *newprog.f***

Comment Messages

These messages indicate valid but unadvisable use of the language being compiled. The compiler displays comments by default. You can suppress comment messages with:

`-cm`	Suppresses all comment messages.

Comment messages do not terminate translation or linking, they do not interfere with any output files either. Some examples of the comment messages are:

```
Null CASE construct

The use of a non-integer DO loop variable or expression

Terminating a DO loop with a statement other than CONTINUE or ENDDO
```

Error Messages

These messages report syntactic or semantic misuse of Fortran. The compiler always displays error messages. Errors suppress object code for the error containing the error and prevent linking, but they make it possible for the parsing to continue to scan for any other errors. Some representative error messages are:

```
line exceeds 132 characters

unbalanced parenthesis

incomplete string
```

Suppressing or Enabling Error Messages

The error conditions are reported in the various stages of the compilation and at different levels of detail as explained below. For various groups of error messages, see Lists of Error Messages.

`-e90, -e95`	Enables issuing of errors rather than warnings for features that are non-standard Fortran.
`-q`	Suppresses compiler output to standard error, `stderr`. When `-q` is specified in conjunction with `-bd`, then only fatal error messages are output to `stderr` by the binder tool provided with the Intel® Fortran Compiler.
`-d{n}`	Generates extra information needed to produce a list of current variables in a diagnostic report. For more details on `-d{n}`, see Selecting a Postmortem Report, -d{n}. Diagnostic reports are generated by the following: • input-output errors • an invalid reference to a pointer or an allocatable array (if `-CA` option selected) • subscripts out of bounds (if `-CB` option selected) • an invalid array argument to an intrinsic procedure (if `-CS` option selected) • use of unassigned variables (if `-CU` option selected)

	• argument mismatch (if `-CV` option selected) • invalid assigned labels • a call to the abort routine • certain mathematical errors reported by intrinsic procedures • hardware detected errors:

Fatal Errors

These messages indicate environmental problems. Fatal error conditions stop translation, assembly, and linking. If a fatal error ends compilation, the compiler displays a termination message on standard error output. Some representative fatal error messages are:

```
Disk is full, no space to write object file

Incorrect number of intrinsic arguments

Too many segments, object format cannot support this many segments
```

Mixing C and Fortran

This section discusses implementation-specific ways to call C procedures from a Fortran program.

Naming Conventions

By default, the Fortran compiler converts function and subprogram names to lower case, and adds a trailing underscore. The C compiler never performs case conversion. A C procedure called from a Fortran program must, therefore, be named using the appropriate case. For example, consider the following calls:

`CALL PROCNAME()`	The C procedure must be named `procname_`.
`x=fnname()`	The C procedure must be named `fnname_`.

In the first call, any value returned by `procname` is ignored. In the second call to a function, `fnname` must return a value.

Passing Arguments between Fortran and C Procedures

By default, Fortran subprograms pass arguments by reference; that is, they pass a pointer to each actual argument rather than the value of the argument. C programs, however, pass arguments by value. Consider the following:

- When a Fortran program calls a C function, the C function's formal arguments must be declared as pointers to the appropriate data type.
- When a C program calls a Fortran subprogram, each actual argument must be specified explicitly as a pointer.

Using Fortran Common Blocks from C

When C code needs to use a common block declared in Fortran, an underscore (_) must be appended to its name, see below.

```
Fortran code
common /cblock/ a(100)
  real a
```

C code

```
struct acstruct {
float a[100];
};
extern struct acstruct
cblock_;
```

Example

This example demonstrates defining a `COMMON` block in Fortran for Linux, and accessing the values from C.

Fortran code

```
COMMON /MYCOM/ A, B(100),I,C(10)
REAL(4) A
REAL(8) B
INTEGER(4) I
COMPLEX(4) C
A = 1.0
B = 2.0D0
I = 4
C = (1.0,2.0)
CALL GETVAL()
END
```

C code

```
typedef struct compl  complex;
struct compl{
float real;
float imag;
};

extern struct {
float a;
double b[100];
int i;
complex c[10];
} mycom_;

void getval_(){
printf("a = %f\n",mycom_.a);
printf("b[0] = %f\n",mycom_.b[0]);
printf("i = %d\n",mycom_.i);
printf("c[1].real = %f\n",mycom_.c
[1].real);
}

penfold%  ifc common.o getval.o -o
common.exe
```

```
penfold%  common.exe
a = 1.000000
b[0] = 2.000000
i = 4
c[1].real = 1.000000
```

Fortran and C Scalar Arguments

Table that follows shows a simple correspondence between most types of Fortran and C data.

Fortran and C Language Declarations

Fortran	C
integer*1 x	char x;
integer*2 x	short int x;
integer*4 x	long int x;
integer x	long int x;
integer*8 x	long long x; or _int64 x;
logical*1 x	char x;
logical*2 x	short int x;
logical*4x	long int x;
logical x	long int x;
logical*8 x	long long x; or _int64 x;
real*4 x	float x;
real*8 x	double x;
real x	float x;
real*16	No equivalent
double precision x	double x;
complex x	struct {float real, imag;} x;
complex*8 x	struct {float real, imag;} x;
complex*16 x	struct {double dreal, dimag;} x;
double complex x	struct {double dreal, dimag;} x;
complex(KIND=16)x	No equivalent
character*6 x	char x[6];

Example below illustrates the correspondence shown in the table above: a simple Fortran call and its corresponding call to a C procedure. In this example the arguments to the C procedure are declared as pointers.

Example of Passing Scalar Data Types from Fortran to C

Fortran Call

```
integer I
integer*2 J
real x
double precision d
logical l
call vexp( i, j, x, d, l )
```

C Called Procedure

```
void vexp_ ( int *i, short *j, float
*x, double *d, int *l )
{
...program text...
}
```

Note

The `character` data or `complex` data do not have a simple correspondence to C types.

Passing Scalar Arguments by Value

A Fortran program compiled with the Intel® Fortran Compiler can pass scalar arguments to a C function by value using the nonstandard built-in function %VAL. The following example shows the Fortran code for passing a scalar argument to C and the corresponding C code.

Example of Passing Scalar Arguments from Fortran to C

Fortran Call

```
integer i
double precision f, result,
argbyvalue
result= argbyvalue(%VAL(I),%VAL
(F))
END
```

C Called Function

```
double argbyvalue_ (int i,double
f)
{
...program text...
return g;
}
```

In this case, the pointers are not used in C. This method is often more convenient, particularly to call a C function that you cannot modify, but such programs are not always portable.

Note

Arrays, records, `complex` data, and `character` data cannot be passed by value.

Array Arguments

The table below shows the simple correspondence between the type of the Fortran actual argument and the type of the C procedure argument for arrays of types `INTEGER`, `INTEGER*2`, `REAL`, `DOUBLE PRECISION`, and `LOGICAL`.

Note
There is no simple correspondence between Fortran automatic, allocatable, adjustable, or assumed size arrays and C arrays. Each of these types of arrays requires a Fortran array descriptor, which is implementation-dependent.

Array Data Type

Fortran Type	C Type
`integer x( )`	`int x[ ];`
`integer*1 x( )`	`signed char x[ ];`
`integer*2 x( )`	`short x[ ];`
`integer*4 x( )`	`long int x[ ];`
`integer*8 x( )`	`long long x[ ];` or `_int64`
`real*4 x( )`	`float x[ ];`
`real*8 x( )`	`double x[ ];`
`real x( )`	`float x[ ];`
`real*16 x( )`	No equivalent
`double precision x ( )`	`double x[ ];`
`logical*1 x( )`	`char x[ ];`
`logical*2 x( )`	`short int x[ ];`
`logical*4 x( )`	`long int x[ ];`
`logical x( )`	`int x[ ];`
`logical*8 x( )`	`long long x[ ];` or `_int64 x [ ];`
`complex x( )`	`struct {float real, imag;} [x];`
`complex *8 x( )`	`struct {float real, imag;} [x];`
`complex *16 x( )`	`struct {double dreal,dimag;} x;`
`double complex x( )`	`struct { double dreal,dimag; } [x];`
`complex(KIND=16) x ( )`	No equivalent

Note
Be aware that array arguments in the C procedure do not need to be declared as pointers. Arrays are always passed as pointers.

Note
When passing arrays between Fortran and C, be aware of the following semantic differences:

- Fortran organizes arrays in column-major order (the first subscript, or dimension, of a multiply-dimensioned array varies the fastest); C organizes arrays in row-major order (the last dimension varies the fastest).
- Fortran array indices start at 1 by default; C indices start at 0. Unless you declare the Fortran array with an explicit lower bound, the Fortran element `X(1)` corresponds to the C `element x[0]`.

Example below shows the Fortran code for passing an array argument to C and the corresponding C code.

Example of Array Arguments in Fortran and C

```
Fortran Code
dimension i(100), x(150)
call array( i, 100, x, 150 )
```

```
Corresponding C Code
array ( i, isize, x, xsize )
int i[ ];
float x[ ];
int *isize, *xsize;
{
. . .program text. . .
}
```

Character Types

If you pass a `character` argument to a C procedure, the called procedure must be declared with an extra integer argument at the end of its argument list. This argument is the length of the `character` variable.

The C type corresponding to `character` is `char`. Example that follows shows Fortran code for passing a character type called `charmac` and the corresponding C procedure.

Example of Character Types Passed from Fortran to C

```
Fortran Code
character*(*) c1
character*5 c2
float x
call charmac( c1, x, c2 )
```

```
Corresponding C Procedure
charmac_ (c1, x, c2, n1, n2)
int n1, n2;
char *c1,*c2;
float *x;
{
. . .program text. . .
}
```

For the corresponding C procedure in the above example, `n1` and `n2` are the number of characters in `c1` and `c2`, respectively. The added arguments, `n1` and `n2`, are passed by value, not by reference. Since the string passed by Fortran is not null-terminated, the C procedure must use the length passed.

Null-Terminated CHARACTER Constants

As an extension, the Intel Fortran Compiler enables you to specify null-terminated `character` constants. You can pass a null-terminated character string to C by making the length of the `character` variable or array element one character longer than otherwise necessary, to provide for the null character. For example:

Fortran Code

```
PROGRAM PASSNULL

interface
subroutine croutine (input)
!MS$attributes alias:'-
croutine'::CROUTINE
character(len=12) input
end subroutine
end interface

character(len=12)HELLOWORLD
data_HELLOWORLD/'Hello World'C/
call croutine(HELLOWORLD)
end
```

Corresponding C Code

```
void croutine(char *input, int len)
{
printf("%s\n",input);
}
```

Complex Types

To pass a `complex` or `double complex` argument to a C procedure, declare the corresponding argument in the C procedure as either of the two following structures, depending on whether the actual argument is `complex` or `double complex`:

```
struct { float real, imag; } *complex;
```

```
struct { double real, imag; } *dcomplex;
```

Example below shows Fortran code for passing a complex type called `compl` and the corresponding C procedure.

Example of Complex Types Passed from Fortran to C

Fortran Code

`double complex dc`
`complex c`
`call compl( dc, c)`

Corresponding C Procedure

`compl ( dc, c )`
`struct { double real, imag; } *dc;`
`struct { float real, imag; } *c;`
`{`
`. . .program text. . .`
`}`

Return Values

A Fortran subroutine is a C function with a void return type. A C procedure called as a function must return a value whose type corresponds to the type the Fortran program expects (except for `character`, `complex`, and `double complex` data types). The table below shows this correspondence.

Return Value Data Type

Fortran Type	C Type
`integer`	`int;`
`integer*1`	`signed char;`
`integer*2`	`short;`
`integer*4`	`long int x;`
`integer*8 x`	`long long x;` or `_int64`
`logical`	`int;`
`logical*1`	`char;`
`logical*2`	`short;`
`logical*4x`	`long int x;`
`logical*8`	`long long x;` or `_int64`
`real`	`float;`
`real*r x`	`float x;`
`real*8 x`	`double x;`
`real*16`	No equivalent
`double precision`	`double;`

Example below shows Fortran code for a return value function called `cfunct` and the corresponding C routine.

Example of Returning Values from C to Fortran

```
Fortran code
integer iret, cfunct
iret = cfunct()
```

```
Corresponding C Routine
int cfunct ()
{
...program text...
return i;
}
```

Returning Character Data Types

If a Fortran program expects a function to return data of type `character`, the Fortran compiler adds two additional arguments to the beginning of the called procedure's argument list:

- The first argument is a pointer to the location where the called procedure should store the result.
- The second is the maximum number of characters that must be returned, padded with white spaces if necessary.

The called routine must copy its result through the address specified in the first argument. Example that follows shows the Fortran code for a return character function called `makechars` and corresponding C routine.

Example of Returning Character Types from C to Fortran

```
Fortran code
character*10 chars, makechars
double precision x, y
chars = makechars( x, y )
```

```
Corresponding C Routine
void makechars_ ( result, length, x,
y );
char *result;
int length;
double *x, *y;
{
...program text, producing
returnvalue...
for (i = 0;  i < length;  i++ ) {
result[i] = returnvalue[i];
```

```
}
}
```

In the above example, the following restrictions and behaviors apply:

- The function's `length` and `result` do not appear in the call statement; they are added by the compiler.
- The called routine must copy the `result` string into the location specified by `result`; it must not copy more than `length` characters.
- If fewer than `length` characters are returned, the return location should be padded on the right with blanks; Fortran does not use zeros to terminate strings.
- The called procedure is type `void`.
- You must use lowercase names for C routines or ATTRBUTE directives and `INTERFACE` blocks to make the calls using uppercase.

Returning Complex Type Data

If a Fortran program expects a procedure to return a `complex` or `double-complex` value, the Fortran compiler adds an additional argument to the beginning of the called procedure argument list. This additional argument is a pointer to the location where the called procedure must store its result.

Example below shows the Fortran code for returning a complex data type procedure called `wbat` and the corresponding C routine.

Example of Returning Complex Data Types from C to Fortran

Fortran code

```
complex bat, wbat
real x, y
bat = wbat ( x, y )
```

Corresponding C Routine

```
struct _mycomplex { float real, imag };
typedef struct _mycomplex _single_complex;
void wbat_ (_single_complex location, float
*x, float *y)

{
float realpart;
float imaginarypart;
... program text, producing realpart and
imaginarypart...
*location.real = realpart;
*location.imag = imaginarypart;
}
```

In the above example, the following restrictions and behaviors apply:

- The argument location does not appear in the Fortran call; it is added by the compiler.
- The C subroutine must copy the result's real and imaginary parts correctly into `location`.
- The called procedure is type `void`.

If the function returned a `double complex` value, the type `float` would be replaced by the type `double` in the definition of location in `wbat`.

Procedure Names

C language procedures or external variables can conflict with Fortran routine names if they use the same names in lower case with a trailing underscore. For example:

Fortran Code

```
subroutine myproc(a,b)
end
```

C Code

```
void myproc_( float *a, float *b){
}
```

The expressions above are equivalent, but conflicting routine declarations. Linked into the same executable, they would cause an error at link time.

Many routines in the Fortran runtime library use the naming convention of starting library routine names with an `f_` prefix. When mixing C and Fortran, it is the responsibility of the C program to avoid names that conflict with the Fortran runtime libraries.

Similarly, Fortran library procedures also include the practice of appending an underscore to prevent conflicts.

Pointers

In the Intel® Fortran Compiler implementation, pointers are represented in memory in the form shown in the table that follows.

Pointer Representation in Intel Fortran Compiler

Pointer To:	Representation
a numeric scalar	one word representing the address of its target
a derived type scalar	one word representing the address of its target
a character scalar	two words, the first word containing the address of its target and the second containing its defined length
an array	a data structure of variable size that describes the target array; Intel reserves the right to modify the form of this structure without notice

Calling C Pointer-type Function from Fortran

In Intel® Fortran, the result of a C pointer-type function is passed by reference as an additional, hidden argument. The function on the C side needs to emulate this as follows:

Calling C Pointer Function from Fortran

Fortran code

```
program test
interface
function cpfun()
integer, pointer:: cpfun
end function
end interface
integer, pointer:: ptr
ptr => cpfun()
print*, ptr
end
```

C Code

```
#include <malloc.h>
void *cpfun_(int **LP)
{
*LP = (int *)malloc(sizeof
(int));
**LP = 1;
return LP;
}
```

The function's result (`int *`) is returned as a pointer to a pointer (`int **`), and the C function must be of type `void` (`not int*`). The hidden argument comes at the end of the argument list, if there are other arguments, and after the hidden lengths of any character arguments.

In addition to pointer-type functions, the same mechanism should be used for Fortran functions of user-defined type, since they are also returned by reference as a hidden

argument. The same is true for functions returning a derived type (`structure`) or `character` if the function is `character*(*)`.

Note
Calling conventions such as these are implementation-dependent and are not covered by any language standards. Code that is using them may not be portable.

Implicit Interface

An implicit interface call is a call on a procedure in which the caller has no explicit information on the form of the arguments expected by the procedure; all calls within a Fortran program are of this form. All arguments passed through an implicit interface, apart from label arguments, are passed by address.

Fortran Implicit Argument Passing by Address

Argument	Address Passed
scalar	the address of the scalar
array	the address of the first element of the array
scalar pointer	the address of its target
array pointer	the address of the first element of its target
procedure	the address associated with the external name

Actual arguments of type `character` are passed as a character descriptor, which consists of two words, see Character Types.

Label arguments (alternate returns) are handled differently: subroutines which include one or more alternate returns in the argument list are compiled as integer functions; these functions return an index into a computed `goto`; the caller executes these `goto`s on return. For example:

```
call validate(x,*10,*20,*30)
```

is equivalent to

```
goto (10,20,30), validate(x)
```

Explicit Interface

Fortran provides various mechanisms by which the declarations of the dummy arguments within the called procedure can be made available to the caller while it is constructing the actual argument list. An explicit interface call is one to the following:

- a module procedure
- an internal procedure

- an external procedure for which an interface block is provided

In this form of call the construction of the actual argument list is controlled by the declarations of the dummy arguments, rather than by the characteristics of the actual arguments. As in an implicit interface call, all arguments (apart from label arguments) are passed by address, but the form of the address is controlled by attributes of the associated dummy argument, see the table below.

Fortran Explicit Argument Passing by Address

Argument	Address Passed
scalar	the address of the scalar
assumed-shape array	the address of an internal data structure which describes the actual argument
other arrays	the address of the first element of the actual array
scalar pointer	the address of the pointer
array pointer	the address of an internal data structure which describes the pointer's target
procedure	the address associated with the external name

As in an implicit interface call, arguments of type `character` are passed as a character descriptor, described in Character Types.

Intel reserves the right to alter or modify the form of the internal data used to pass assumed-shape arrays and pointers to arrays. It is therefore not recommended that interfaces using these forms of argument are to be compiled with other than Intel® Fortran Compiler.

The call on an explicit interface need not associate an actual argument with a dummy argument if the dummy argument has the `optional` attribute. An `optional` argument that is not present for a particular call to a routine has a placeholder value passed instead of its address. The place-holder value for optional arguments is always -1.

Intrinsic Functions

The normal argument passing mechanisms described in the preceding sections may sometimes not be appropriate when calling a procedure written in C. The Intel® Fortran Compiler also provides the intrinsic functions %REF and %VAL which may be used to modify the normal argument passing mechanism. These intrinsics must not be used when calling a procedure compiled by the Intel Fortran Compiler. See Additional Intrinsic Functions section.

Reference Information

Maximum Size and Number

The table below shows the size or number of each item that the Intel® Fortran Compiler can process. All capacities shown in the table are tested values; the actual number can be greater than the number shown.

Item	Tested Values
Maximum nesting of interface blocks	10
Maximum nesting of input/output implied DOs	20
Maximum nesting of array constructor implied DOs	20
Maximum nesting of include files	10
Maximum length of a character constant	32767
Maximum Hollerith length	4096
Maximum number of digits in a numeric constant	1024
Maximum nesting of parenthesized formats	20
Maximum nesting of DO, IF or CASE constructs	100
Maximum number of arguments to MIN and MAX	255
Maximum number of parameters	256
Maximum number of continuation lines in fixed or free form	99
Maximum width field for a numeric edit descriptor	1024

Additional Intrinsic Functions

The Intel® Fortran Compiler provides a few additional generic functions, and adds specific names to standard generic functions (in particular, to accommodate DOUBLE COMPLEX arguments). Some specific names are synonyms to standard names.

Note
Many intrinsics listed in this section are handled as library calls. Not all the functions that are listed in the sections that follow can be inlined.

Synonyms

The Intel® Fortran provides synonyms for standard Fortran intrinsic names. They are given in the right-hand columns.

Standard Name	Intel Fortran Synonym	Standard Name	Intel Fortran Synonym
DBLE	DREAL	DIGITS	EPPREC
IAND	AND	MINEXPONENT	EPEMIN
IEOR	XOR	MAXEXPONENT	EPEMAX
IOR	OR	HUGE	EPHUGE
RADIX	EPBASE	EPSILON	EPMRSP

Note that the Fortran standard intrinsic TINY and the Intel additional intrinsic EPTINY are not synonyms. TINY returns the smallest positive normalized value appropriate to the type of its argument, whereas EPTINY returns the smallest positive denormalized value.

DCMPLX Function

The DCMPLX function must satisfy the following conditions:

- If x is of type DOUBLE COMPLEX, then DCMPLX(x) is x.
- If x is of type INTEGER, REAL, or DOUBLE PRECISION, then DCMPLX(x) is DBLE(x) + 0i
- If x1 and x2 are of type INTEGER, REAL or DOUBLE PRECISION, then DCMPLX(x1, x2) is

  DBLE(x1) + DBLE(x2) * i
- If DCMPLX has two arguments, then they must be of the same type, which must be INTEGER, REAL or DOUBLE PRECISION.
- If DCMPLX has one argument, then it may be INTEGER, REAL or DOUBLE PRECISION, COMPLEX or DOUBLE COMPLEX.

LOC Function

The LOC function returns the address of a variable or of an external procedure.

Intel® Fortran KIND Parameters

Each intrinsic data type (INTEGER, REAL, COMPLEX, LOGICAL and CHARACTER) has a KIND parameter associated with it. The actual values which the KIND parameter for each intrinsic type can take are implementation-dependent. The Fortran standard specifies that these values must be INTEGER, that there must be at least two REAL KINDs and two COMPLEX KINDs (corresponding in each case to default REAL and DOUBLE PRECISION), and that there must be at least one KIND for each of the INTEGER, CHARACTER and LOGICAL data types.

INTEGER KIND values

KIND=1 1-byte INTEGER
KIND=2 2-byte INTEGER
KIND=4 4-byte INTEGER *default* KIND
KIND=8 8-byte INTEGER

REAL KIND values

KIND=4 4-byte REAL *default* KIND
KIND=8 8-byte REAL *equivalent to* DOUBLE PRECISION
KIND=16 16-byte REAL

COMPLEX KIND values

KIND=4 4-byte REAL & imaginary parts *default* KIND
KIND=8 8-byte REAL & imaginary parts *equivalent to* DOUBLE COMPLEX
KIND=16 16-byte REAL and imaginary parts *equivalent to* COMPLEX*32

LOGICAL KIND values

KIND=1 1-byte LOGICAL
KIND=2 2-byte LOGICAL
KIND=4 4-byte LOGICAL *default* KIND
KIND=8 8-byte LOGICAL

CHARACTER KIND value

KIND=1 1-byte CHARACTER *default* KIND

Except for COMPLEX, the KIND numbers match the size of the type in bytes. For COMPLEX the KIND number is the KIND number of the REAL or imaginary part.

An include file (f90_kinds.f90) providing symbolic definitions, for use when defining KIND type parameters, is included as part of the standard Intel® Fortran release.

Argument and Result KIND Parameters

The following extensions to standard Fortran are provided:

- References to the following intrinsic functions return INTEGER(KIND=2) results when compile-time option -I2 or -i2 is specified: INT, IDINT, NINT, IDNINT, IFIX, MAX1, MIN1.
- The following specific intrinsic functions may be given arguments of type INTEGER (KIND=2): IABS, FLOAT, MAX0, AMAX0, MIN0, AMIN0, IDIM, ISIGN.
- References to the following intrinsic functions return INTEGER(KIND=8): results when compile-time option -I2 or -i2 is specified: INT, IDINT, NINT, IDNINT, IFIX, MAX1, MIN1.
- The following specific intrinsic functions may be given arguments of type INTEGER (KIND=8): IABS, FLOAT, MAX0, AMAX0, MIN0, AMIN0, IDIM, ISIGN.
- References to the following specific intrinsic functions return REAL(KIND=8) results

when compile-time option -r8 is specified: ALOG, ALOG10, AMAX1, AMIN1, AMOD, MAX1, MIN1, SNGL, REAL.

- References to the following specific intrinsic functions return results of type COMPLEX(KIND=8), that is the real and imaginary parts are each of 8 bytes, when compile-time option -r8 is specified: CABS, CCOS, CEXP, CLOG, CSIN, CSQRT, CMPLX.

%REF and %VAL Intrinsic Functions

Intel® Fortran provides two additional intrinsic functions, %REF and %VAL, that can be used to specify how actual arguments are to be passed in a procedure call. They should not be used in references to other Fortran procedures, but may be required when referencing a procedure written in another programming language such as C.

%REF(X)	Specifies that the actual argument X is to be passed as a reference to its value. This is how Intel Fortran normally passes arguments except those of type character. For each character value that is passed as an actual argument, Intel Fortran normally passes both the address of the argument and its length (with the length being appended on to the end of the actual argument list as a hidden argument. Passing a character argument using %REF does not pass the hidden length argument.
%VAL(X)	Specifies that the value of the actual argument X is to be passed to the called procedure rather than the traditional mechanism employed by Fortran where the address of the argument is passed.

In general, %VAL passes its argument as a 32-bit, sign extended, value with the following exceptions: the argument cannot be an array, a procedure name, a multibyte Hollerith constant, or a character variable (unless its size is explicitly declared to be 1).

In addition, the following conditions apply:

- If the argument is a derived type scalar, then a copy of the argument is generated and the address of the copy is passed to the called procedure.
- An argument of complex type will be viewed as a derived-type containing two fields - a real part and an imaginary part, and is therefore passed in manner similar to derived-type scalars.
- An argument that is a double-precision real will be passed as a 64-bit floating-point value.

This behavior is compatible with the normal argument passing mechanism of the C programming language, and it is to pass a Fortran argument to a procedure written in C where %VAL is typically used.

The intrinsic procedures %REF and %VAL can only be used in each explicit interface block, or in the actual CALL statement or function reference as shown in the example that follows.

Calling Intrinsic Procedures

```
PROGRAM FOOBAR
INTERFACE
SUBROUTINE FRED(%VAL(X))
INTEGER :: X
END SUBROUTINE FRED
FUNCTION FOO(%REF(IP))
INTEGER :: IP, FOO
END FUNCTION FOO
END INTERFACE
...
CALL FRED(I)  ! The value of I is
passed to FRED
J = FOO(I)    ! I passed to FOO by
reference,
! FOO receives a reference to
! the value of I.
END PROGRAM

Alternatively:
PROGRAM FOOBAR
INTEGER :: FOO
EXTERNAL FOO, FRED
CALL fred(%VAL(I))
J = FOO(%REF(I))
END PROGRAM
```

List of Additional Intrinsic Functions

To understand the tabular list of additional intrinsic functions that follows after these notes, take into consideration the following:

- Specific names are only included in the Additional Intrinsic Functions table if they are not part of standard Fortran.
- An intrinsic that takes an integer argument accepts either INTEGER(KIND=2) or INTEGER(KIND=4) or INTEGER(KIND=8).
- The abbreviation "double" stands for DOUBLE PRECISION.
- The abbreviation "dcomplex" stands for DOUBLE COMPLEX. Dcomplex type is an Intel® Fortran extension, as are all intrinsic functions taking dcomplex arguments or returning dcomplex results.
- If an intrinsic function has more than one argument, then they must all be of the same type.