Intel® Fortran Compiler
User's Guide

Cpopyright 1996 - 2002 Intel Corporation
All rights reserved
Issued in USA
Document No. FL-600-06

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.

Except as permitted by such license, no part of this document may be reproduced, stored in a retrieval system, or transmitted in any form or by any means without the express written consent of Intel Corporation.

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.

Intel, Pentium, Pentium Pro, Itanium, MMX, Celeron, 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.

Copyright © Intel Corporation 1996 - 2002.

Table of Contents

ABOUT INTEL® FORTRAN COMPILER 13

Welcome to Intel® Fortran Compiler 13
Major Components of the Intel® Fortran Compiler Product 13
What's New in This Release 13
Compiler for Two Architectures 13
Improvements and New Features 13
Auto-parallelization 14
OpenMP* Support 14
Hyper-Threading Technology support 14
Optimizing for IA-32 Processors 14
Little-endian to Big-endian Conversion 14
Optimizing for Itanium Processor Family 14
Optimization Reports (Itanium(TM) Compiler) 14
Features and Benefits 14
Product Web Site and Support 15
System Requirements 15
Minimum Hardware Requirements 15
Operating System Requirements 16
Browser 16
FLEXlm* Electronic Licensing 16
About This Document 16
How to Use This Document 16
Notation Conventions 17
Related Publications 17
Publications on Compiler Optimizations 18

COMPILER OPTIONS QUICK REFERENCE GUIDES 20

Options Quick Reference Guides Overview 20
Conventions used in the Options Quick Guide Tables 20
New Compiler Options 21
Alphabetical Listing 23
Compiler Options Quick Reference Alphabetical 23
Functional Group Listings 39
Compiler Options by Functional Groups Overview 39
Key to the Tables 39
Customizing Compilation Process Options 39
Fortran Compilation Environment 39
Alternate Tools and Locations 40

Preprocessing 40
Compiling 41
Linking 42
Compilation Output 42
Debugging 43
Libraries 43
Diagnostics and Messages 44
Runtime Diagnostics (IA-32 Compiler only) 44
Compiler Information Messages 45
Comment and Warning Messages 45
Error Messages 45
Language Conformance Options 46
Data Type 46
Source Program 46
Arguments and Variables 47
Common Blocks 48
Application Performance Optimizations Options 49
Setting Optimization Level 49
Floating-point Arithmetic Precision 49
Processor Dispatch Support (IA-32 only) 50
Interprocedural Optimizations 51
Profile-guided Optimizations 52
High-level Language Optimizations 52
Parallelization 52
Vectorization (IA-32 only) 53
Optimization Reports (Itanium(TM) Compiler) 54
Windows* to Linux* Mapping 55
Windows* to Linux* Options Cross-reference 55
GETTING STARTED WITH THE INTEL® FORTRAN COMPILER 72
Invoking Intel Fortran Compiler 72
Invoking from the Compiler Command Line 72
Setting the Environment Variables 72
Command Line Syntax 73
Command Line with make 73
Input Files 74
Default Behavior of the Compiler 74
Default Behavior Overview 74
Default Behavior of the Compiler Options 75
Data Setting and Language Conformance 75
Optimizations 76
Compilation 77
Messages and Diagnostics 77
Disabling Default Options 77
Resetting Default Data Types 78

Default Libraries and Tools 78
Assembler 78
Linker 79
Compilation Phases 79
Application Development Cycle 79
CUSTOMIZING COMPILATION ENVIRONMENT 81
Customizing Compilation Environment Overview 81
Environment Variables 81
Configuration File Environment Variables 82
FCE Options 82
Configuration Files 82
Response Files 83
Include Files 84
Fortran Compilation Environment (FCE) 84
FCE Overview 84
Object Files and Dictionary Files 84
Program Unit Catalog Files 85
Specifying the Name and Path of the PUCLF 85
Guidelines for the PUCLF 85
An Example of Development Organization 86
The FCE Manager Utility 86
The Binder 89
Activating the Binder 90
Advantages of Using the Binder 91
Dependent and Independent Compilation 91
Fortran Programs with or without Modules 91
Small-Scale Projects 91
Larger-Scale Projects 92
Fortran Programs Without Modules 92
Stale Program Units 92
CUSTOMIZING COMPILATION PROCESS 94
Customizing Compilation Process Overview 94
Specifying Alternate Tools and Locations 94
Specifying an Alternate Component (-Qlocation,tool,path) 94
Passing Options to Other Tools (-Qoption,tool,opts) 95
Preprocessing 95
Preprocessing Overview 95
Preprocessor Options 95
Preprocessing Fortran Files 96
Enabling Preprocessing with Compiler Options 97

String Constants for IA-32 Systems....97
Preprocessing Only: -E, -EP, -F, and -P....97
Searching for Include Files....98
Specifying and Removing Include Directory Search: -I, -X....98
Specifying an Include Directory, -I....98
Removing Include Directories, -X....98
Defining Macros, -D, -U and -A....98
Predefined Macros....99
Compiling....100
Compilation Overview....100
Compilation Options....100
Controlling Compilation....100
Controlling Compilation Phases....100
Aliasing....101
Translating Other Code to Fortran....101
Saving Compiler Version and Options Information, -sox....101
Monitoring Data Settings....101
Specifying Structure Tag Alignments....101
Allocation of Zero-initialized Variables, -nobss_init....102
Monitoring Data for IA-32 Systems....102
Correcting Computations for IA-32 Processors, -0f_check....102
The ebp Register Usage....102
Monitoring Data for Itanium(TM)-based Systems....102
Flushing to Zero Denormal Values, -ftz....102
Little-endian-to-Big-endian Conversion (IA-32)....102
Little-to-Big Endian Conversion Environment Variable....102
Examples....103
Specifying Compilation Output....104
Specifying Compilation Output Overview....104
Default Output Files....104
Specifying Executable Files....105
Specifying Object Files....105
Specifying Assembly Files....106
Producing Assembly Files with Annotations and Comments....106
Compiler Output Options Summary....106
Using the Assembler to Produce Object Code....107
Assembly File Code Example....107
Listing Options....109
Linking....109
Options to Link to Tools and Libraries....109
Controlling Linking and its Output....109
Suppressing Linking....109
Debugging....110
Debugging Options Overview....110
Support for Symbolic Debugging....110
Debugging and Assembling....110

Parsing for Syntax Only 110
Compiling Source Lines with Debugging Statements, -DD 111
Debugging and Optimizations 111
-fp Option and Debugging (IA-32 only) 111
FORTRAN LANGUAGE CONFORMANCE OPTIONS 114
Fortran Language Options Overview 114
Setting Integer and Floating-point Data Types 114
Integer Data 114
Floating-point Data 114
Source Program Features 114
Program Structure and Format 114
Compatibility with Platforms and Compilers 114
Escape Characters 115
Line Terminators 115
Setting Arguments and Variables 115
Automatic Allocation of Variables to Stacks 116
Alignment, Aliases, Implicit None 116
Allocating Common Blocks 117
Dynamic Common Option 117
Allocating Memory to Dynamic Common Blocks 117
Why Use a Dynamic Common 118
Rules of Using Dynamic Common Option 118
OPTIMIZATIONS 119
Optimization Levels 119
Optimization Levels Overview 119
Setting Optimization Levels 119
Itanium Compiler 120
IA-32 Compiler 120
IA-32 and Itanium Compilers 120
Restricting Optimizations 121
Floating-point Arithmetic Optimizations 121
Floating-point Arithmetic Precision Overview 121
-mp Option 121
-mp1 Option 121
Floating-point Arithmetic Precision for IA-32 Systems 121
-prec_div Option 122
-pc{32|64|80} Option 122
Rounding Control, -rcd, -fp_port 122
Floating-point Arithmetic Precision for Itanium-based Systems 122
Contraction of FP Multiply and Add/Subtract Operations 122
FP Speculation 123
FP Operations Evaluation 123

Controlling Accuracy of the FP Results 123
Maintaining and Restricting FP Arithmetic Precision 123
Processor Dispatch Extensions Support (IA-32 Only) 124
Targeting a Processor and Extensions Support Overview 124
Targeting a Processor, -tpp{n} 125
Optimizing for a Specific Processor Without Excluding Others 125
Exclusive Specialized Code with -x{i|M|K|W} 125
-x Summary 126
Specialized Code with -ax{i|M|K|W} 126
-ax Summary 126
Checking for Performance Gain 127
Combining Processor Target and Dispatch Options 127
Example of -x and -ax Combinations 128
Interprocedural Optimizations (IPO) 128
IPO Overview 128
Multifile IPO 129
Multifile IPO Overview 129
Compilation Phase 129
Linkage Phase 129
Compilation with Real Object Files, -ipo_obj 129
Creating a Multifile IPO Executable 130
Creating a Multifile IPO Executable Using a Project Makefile 130
Analyzing the Effects of Multifile IPO, -ipo_c, -ipo_S 131
Using -ip witn -Qoption Specifiers 131
-Qoption Specifiers 131
Inline Expansion of Functions 132
Criteria for Inline Function Expansion 132
Controlling Inline Expansion of User Functions 133
Profile-guided Optimizations 134
Profile-guided Optimizations Overview 134
Instrumented Program 134
Added Performance with PGO 134
Profile-guided Optimizations Methodology 134
PGO Phases 134
Basic PGO Options 136
Generating Instrumented Code, -prof_gen[x] 137
Generating a Profile-optimized Executable, -prof_use 137
Advanced PGO Options 137
Specifying the Directory for Dynamic Information Files 137
Specifying Profiling Summary File 137
Guidelines for Using Advanced PGO 137
PGO Environment Variables 138
Example of Profile-Guided Optimization 138
Function Order List 139
Function Order List Overview 139
Usage Guidelines 139

Function Order List Utilities 139
The profmerge Utility 139
The proforder Utility 140
Comparison of Function Order Lists and IPO Code Layout 140
Dump Profile Data Utility 141
Example of Function Order List Generation 141
PGO API: Profile Information Generation Support 142
PGO API Support Overview 142
The Profile IGS Functions 142
The Profile IGS Environment Variable 142
Dumping Profile Information 142
Recommended usage 143
Resetting the Dynamic Profile Counters 143
Recommended usage 143
Dumping and Resetting Profile Information 143
Recommended usage 143
Interval Profile Dumping 143
Recommended usage 144
High-level Language Optimizations (HLO) 144
HLO Overview 144
Loop Transformations 145
Scalar Replacement (IA-32 Only) 145
Loop Unrolling with -unroll[n] 145
Benefits and Limitations of Loop Unrolling 145
Memory Dependency with IVDEP Directive 146
Prefetching 146
Parallelization 147
Parallelization Options Overview 147
Auto-parallelization 147
Enabling Auto-parallelizer 147
Auto-parallelization Environment Variables 148
Guidelines for Effective Auto-parallelization Usage 148
Analyzing Compiler for Auto-parallelization 148
Preparing for Auto-parallelization 148
Threshold for Auto-parallelization 148
Using Auto-parallelization 149
Auto-parallelization Examples 149
Auto-parallelizer's Diagnostic 151
Troubleshooting Tips 151
!DIR$PARALLEL Directive 151
Parallelization with OpenMP* 151
Command Line Options 152
OpenMP* Standard Option 153
OpenMP Fortran Directives and Clauses 153
OpenMP Environment Variables 153
OpenMP* Runtime Library Routines 153

Intel Extensions to OpenMP* 153
Environment Variables 153
Thread-level MALLOC( ) 153
Examples of OpenMP* Usage 154
A Simple Difference Operator 154
Two Difference Operators 154
Vectorization (IA-32 Only) 155
Vectorization Overview 155
Vectorizer Options 155
Vectorization Key Programming Guidelines 156
Guidelines 156
Restrictions 156
Data Dependence 157
Data Dependence Analysis 157
Loop Constructs 158
Loop Exit Conditions 158
Types of Loop Vectorized 160
Stripmining and Cleanup 160
Statements in the Loop Body 160
Floating-point Array Operations 160
Integer Array Operations 161
Other Operations 161
Language Support and Directives 161
IVDEP Directive 161
Overriding Vectorizer's Efficiency Heuristics 162
The VECTOR ALWAYS Directive 162
The VECTOR ALIGNED/UNALIGNED Directives 162
The NOVECTOR Directive 163
Vectorization Examples 163
Argument Aliasing: A Vector Copy 163
Data Alignment 163
Alignment Strategy 164
Loop Interchange and Subscripts: Matrix Multiply 165
Optimizer Report Generation (Itanium(TM) Compiler) 166
Specifying Optimizations to Generate Reports 166
The Availability of Report Generation 167
LIBRARIES 168
Managing Libraries 168
The Order of Passing the Files to Linker 168
Using the POSIX and Portability Libraries 169
Intel® Shared Libraries 169
Advantages of This Approach 169
Shared Library Options 169

Math Libraries 170
Math Libraries Overview 170
Using Math Libraries with IA-32 Systems 171
Optimized Math Library Primitives 171
Programming with Math Library Primitives 172
IEEE Floating-point Exceptions 171
Denormal 172
Divide-by-Zero Exception 172
Overflow Exception 173
Underflow Exception 173
Inexact Exception 173
Invalid Operation Exception 173
DIAGNOSTICS AND MESSAGES 175
Diagnostics Overview 175
Runtime Diagnostics (IA-32 Compiler Only) 175
Runtime Diagnostics Overview 175
Optional Runtime Checks 175
Pointers, -CA 176
Allocatable Arrays 176
Assumed-Shape Arrays 176
Array Subscripts, Character Substrings, -CB 177
Unassigned Variables, -CU 177
Notes on Variables 177
Actual to Dummy Argument Correspondence, -CV 177
Generating Diagnostic Reports 178
Diagnostic Report, -d{n} 178
Selecting a Postmortem Report 179
Invoking a Postmortem Report 179
Postmortem Report Conventions 179
Example 179
Messages and Obtaining Information 181
Compiler Information Messages 181
Diagnostic Messages 181
Command-line Diagnostics 181
Language Diagnostics 182
Warning Messages 182
Suppressing or Enabling Warning Messages 182
Comment Messages 183
Error Messages 183
Suppressing or Enabling Error Messages 184
Fatal Errors 184
MIXING C AND FORTRAN 185
Mixing C and Fortran Overview 185

Naming Conventions 185
Passing Arguments between Fortran and C Procedures 185
Using Fortran Common Blocks from C 185
Example 186
Fortran and C Scalar Arguments 187
Fortran and C Language Declarations 187
Example of Passing Scalar Data Types from Fortran to C 187
Passing Scalar Arguments by Value 188
Example of Passing Scalar Arguments from Fortran to C 188
Array Arguments 188
Array Data Type 189
Example of Array Arguments in Fortran and C 189
Character Types 190
Example of Character Types Passed from Fortran to C 190
Null-Terminated CHARACTER Constants 190
Complex Types 191
Example of Complex Types Passed from Fortran to C 191
Return Values 192
Return Value Data Type 192
Example of Returning Values from C to Fortran 192
Returning Character Data Types 192
Example of Returning Character Types from C to Fortran 193
Returning Complex Type Data 193
Example of Returning Complex Data Types from C to Fortran 194
Procedure Names 194
Pointers 195
Pointer Representation in Intel Fortran Compiler 195
Calling C Pointer-type Function from Fortran 195
Calling C Pointer Function from Fortran 185
Implicit Interface 196
Fortran Implicit Argument Passing by Address 196
Explicit Interface 196
Fortran Explicit Argument Passing by Address 197
Intrinsic Functions 197
REFERENCE INFORMATION 198
OpenMP* Reference Information 198
List of OpenMP* Standard Directives and Clauses 198
OpenMP* Directives 198
OpenMP Clauses 199
List of OpenMP* Runtime Library Routines 199

Compiler Limits 201
Maximum Size and Number 201
Additional Intrinsic Functions 202
Additional Intrinsic Functions Overview 202
Synonyms 202
DCMPLX Function 202
LOC Function 203
Argument and Result KIND Parameters 204
Intel® Fortran KIND Parameters 203
%REF and %VAL Intrinsic Functions 204
List of Additional Intrinsic Functions 205
Intel Fortran Compiler Key Files 208
Key Files Summary for IA-32 Compiler 208
/bin Files 208
/lib Files 209
Key Files Summary for Itanium(TM) Compiler 210
/bin Files 210
/lib Files 211
Lists of Error Messages 213
Error Message Lists Overview 213
Runtime Errors (IA-32 Only) 213
Allocation Errors 215
Input/Output Errors 216
Little-Big Endian Conversion Errors 221
Other Errors Reported by I/O statements 221
Notes 221
Intrinsic Errors 222
Intrinsic Procedure Errors 222
Mathematical Errors 223
Exception Messages 223

About Intel® Fortran Compiler

Welcome to Intel® Fortran Compiler

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

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(TM) Compiler for Itanium-based Applications

The Intel Fortran Compiler for Itanium-based applications includes Intel® Itanium(TM) Assembler and Intel Itanium(TM) 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

Compiler for Two Architectures

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(TM) processor.

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(TM) Compiler for Itanium(TM)-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

- Code size optimization with `-O1`
- Compilation time with `-O2`
- High-level Language optimizations with `-O3`

For new options in this release, see New Compiler Options.

Auto-parallelization

The `-parallel` option detects parallel loops capable of being executed safely in parallel and automatically generates multithreaded code for these loops. Automatic parallelization relieves the user from having to deal with the low-level details of iteration partitioning, data sharing, thread scheduling and synchronizations. It also provides the benefit of the performance available from multiprocessor systems.

OpenMP* Support

The Intel® Fortran Compiler supports OpenMP API version 1.1 and performs code transformation for shared memory parallel programming. The OpenMP support and auto-parallelization are accomplished with the `-openmp` options.

Hyper-Threading Technology support

Both auto-parallelizationa 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.

Optimizing for IA-32 Processors

The `-xW` or `-axW` compiler options generate Streaming SIMD Extensions 2 designed to execute on a Pentium® 4 and Xeon(TM) processor system, see Processor Dispatch Extensions Support. These options improve the performance of applications using processor-specific optimizations, which take advantage of the each processor's architecture features. The inclusion of further vectorizable loops such as threaded loops and search loops improves vectorization results. The `IVDEP` and `VECTOR` directives enhance vectorization.

Little-endian to Big-endian Conversion

The Intel Fortran Compiler can now write unformatted sequential files in big-endian format and read files produced in big-endian format. For details see Little-endian-to-Big-endian Conversion.

Optimizing for Itanium Processor Family

The Itanium architecture provides explicit parallelism, predication, speculation and other features to bring up performance to even higher results. The architecture is highly scalable to fulfill high performance server and workstation requirements.

New option `-ivdep_parallel` enhances performance with the new `IVDEP` directive.

The floating-point arithmetic options (`-ipf_`... series) for Itanium-based applications enable compiler to control optimizations for floating-point computations.

Optimization Reports

For IA-32 applications, levels 4 and 5 have been added to the vectorization reports with `-vec_report`.

For the Itanium-based applications, the `-opt_report` option and its varieties enable to generate optimization reports of different detail level for a number of optimizations.

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 modules 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 documentation web site where you will find links to:

- Fortran compiler home page
- Fortran compiler performance-related topics
- Marketing information
- Related topics on the http://developer.intel.com web site

For internet-based support and resources visit http://developer.intel.com/go/compilers.

For specific details on the Itanium architecture, visit the web site at http://developer.intel.com/design/ia-64/index.htm.

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 and Cross Compiler

- A system based on a Pentium®, Pentium® Pro, Pentium® with MMX(TM) technology, Pentium® II, Pentium® III, Pentium® 4 or Xeon(TM) processor.
- 128 MB RAM
- 100 MB of disk space

Recommended: A system with Pentium® III, Pentium 4 or Xeon processor and 256 MB of RAM

Itanium(TM) Compiler

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

Operating System Requirements

IA-32 architecture:
Linux system with glibc 2.2.2 or 2.2.4 and kernel 2.4. The compiler has been validated with Red Hat Linux* versions 7.1 and 7.2.

Itanium(TM) architecture:
Linux system with glibc 2.2.2 or 2.2.3 and kernel 2.4. The compiler has been validated with Red Hat Linux version 7.1 for Intel Itanium-based systems.

To run Itanium(TM)-based applications you must have an Intel® Itanium(TM) architecture system running the Itanium(TM)-based operating system from RedHat Linux* 7.1. Itanium(TM)-based systems are shipped with all of the hardware necessary to support this product.

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`.

About This Document

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
- VTune® 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(TM) Architecture Application Developer's Architecture Guide*
- *Intel® Itanium(TM) 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(TM) Architecture Software Conventions & Runtime Architecture Guide*
- *Intel® Itanium(TM) Architecture Assembly Language Reference Guide*
- *Intel® Itanium(TM) 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.

Compiler Options Quick Reference Guides

Options Quick Reference Guides Overview

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(TM) 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 diffrerent 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(TM) architecture (Itanium(TM)-based systems only)
- Options available for both IA-32 and Itanium architecture

Option	Description	Default
`-falias`	Enables aliasing in program. More...	ON
`-fno-alias`	Disables aliasing in program. More...	OFF
`-ffnalias`	Enables aliasing across functions. More...	ON
`-fno-fnalias`	Disables aliasing across functions, but enables aliasing across calls. More...	OFF
`-fcode-asm`	Produces assembly file with optional code byte information More...	OFF
`-fsource-asm`	Produces assembly file with optional high-level source code annotations. More...	OFF
`-fverbose-asm`	Produces assembly file with compiler comments including compiler version and options used. More...	ON
`-fnoverbose-asm`	Produces assembly file without compiler comments. More...	OFF
`-ftz` Itanium compiler	Flushes denormal results to zero. 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 fp operations in one of the following modes: *`fast`*: speculate on fp	`-IPF_fp_speculation` *`fast`*

	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_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 the compiler to apply optimizations that affect floating-point accuracy. By default, the compiler may apply optimizations that affect floating-point accuracy. -IPF_fltacc- disables such optimizations. -IPF_fltacc- is effective when -mp is on. More...	ON
-ivdep_parallel Itanium compiler	Indicates there is absolutely no loop-carried memory dependency in the loop where IVDEP directive is specified. More...	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	OFF

	optimizations. More...	
-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
`-parallel`	Enables the auto-parallelizer to generate multithreaded code for loops that can be safely executed in parallel. More...	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. More...	n=75
`-par_report {0|1|2|3}`	Controls the auto-parallelizer's diagnostic levels. More...	`-par_ report1`

Alphabetical Listing

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(TM) architecture (Itanium(TM)-based systems only)
- Options available for both IA-32 and Itanium architecture

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
-A-	Removes all predefined macros. Issues a warning if OpenMP does not work correctly. More...	OFF
-align	Analyzes and reorders memory layout for variables and arrays. (Same as -Zp{n}.) More... To disable, use the -noalign option (default is OFF)	ON
-ansi[-]	Enables (default) or disables assumption of the programs ANSI conformance. More...	ON
-auto	Causes all variables 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. 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 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 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	Dynamically links libraries at run time. Compared to static linking, results in smaller executables.	OFF
-bd,*progname*	Enables the Intel® Fortran Compiler binder to generate a list of objects to build a PROGNAME. More...	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 pointers and allocatable array references for nil. 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	OFF

	conjunction with `-d{n}`. More...	
`-cerrs[-]`	Enables/disables errors and warning messages to be printed in a terse format for diagnostic messages. More...	OFF
`-cl,`*`file`*	Specifies a program unit catalog list file in which to search for referenced modules. 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...	OFF
`-D`*`name`*`[={#|text}]`	Defines a macro name and associates it with the specified value. More...	OFF
`-doubletemps`	Ensures that all intermediate results of floating-point expressions are maintained in at least double	OFF

	precision. More...	
-dps, -nodps	Enable (default) or disable DEC* parameter statement recognition. More...	-dps
-dryrun	Show driver tool commands but do not execute tools. 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 -fpp{n}. 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	Enables aliasing in program. More...	ON
-fno-alias	Disables aliasing in program. More...	OFF
-ffnalias	Enables aliasing across functions. More...	ON
-fno-fnalias	Disables aliasing across functions, but enables aliasing across calls. More...	OFF
-fcode_asm	Produces assembly file with optional code byte annotations. More...	OFF
-fsource_asm	Produces assembly file with optional high-level source code annotations. More...	OFF
-fverbose-asm	Produces assembly file with compiler comments including compiler version and options. More...	ON

-fnoverbose-asm	Produces assembly file without compiler comments. 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}	Runs the Fortran preprocessor (fpp) on all Fortran source files (.f, .ftn, .for, and .f90 files) prior to compilation.
	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.
	More...
		-fpp1

-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 extensions. 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
-G0	Prints source listing to stdout (typically your terminal screen) with the contents of expanded INCLUDE files. More...	OFF
-G1	Prints a source listing to stdout, without contents of expanded	OFF

	INCLUDE files. More...	
-help	Prints help message. More...	OFF
-i{2|4|8}	Defines the default KIND for integer variables and constants in 2, 4, and 8 bytes. More...	-i4
-ic	Runs independent Fortran compilation without accessing and updating Fortran compilation environment (FCE). More...	OFF
-I*dir*	Specifies an additional directory to search for include files whose names do not begin with a slash (/). More...	OFF
-i_dynamic	Enables to link Intel-provided libraries dynamically. More...	OFF
-implicitnone	Enables the IMPLICIT NONE. 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 fp operations in one of the following modes: *fast*: speculate on fp operations;	-IPF_fp_speculation *fast*

	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_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 the compiler to apply optimizations that affect floating-point accuracy. By default, the compiler may apply optimizations that affect floating-point accuracy. -IPF_fltacc- disables such optimizations. -IPF_fltacc- is effective when -mp is on. More...	ON
-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. More...	OFF
-L*dir*	Instructs linker to search *dir* for libraries. More...	OFF
-l*name*	Links with a library indicated in *name*. More...	OFF
-lowercase	Controls the case of routine names and external linker symbols to all lowercase characters. More...	ON
-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, -O2 IA-32 compiler	Optimize for speed. but disable some optimizations that increase code size for a small speed benefit. Disable option -fp. More...	OFF
-O1 Itanium compiler	-O1 turns off software pipelining to reduce code size. Optimizes to favor code size. Enables the same optimizations as -O2 except for loop unrolling. More...	OFF
-O2	Optimizes for speed, but disables some optimizations that increase code size for a small speed benefit. Default. 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
-o*file*	Indicates the executable file name in *file*; for example, -o*myfile*. 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. 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 is ON. More...	OFF

-openmp_ report{0|1|2}	Controls the OpenMP parallelizers diagnostic levels.	-openmp _report1
-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	Enforces 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	n=75

	the loop in parallel, n=0 to 100. More...	
-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
-posixlib	Enables linking to the POSIX library (libPOSF90.a) in the compilation. More...	OFF
-prec_div IA-32 compiler	Improves precision of floating-point divides. Some 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{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) More...	`-r4`
`-rcd` IA-32 compiler	Enables efficient float-to-integer conversions. More...	OFF
`-S`	Produces an assembly output. More...	OFF
`-save`	Saves all variables (static allocation). Opposite of `-auto`. More...	ON
`-scalar_rep[-]` IA-32 compiler	Enables or disables scalar replacement performed during loop transformations (requires `-O3`). More...	OFF
`-sox[-]` IA-32 compiler	Enables (default) or disables saving of compiler options and version in the executable. Itanium compiler: accepted for compatibility only. More...	IA-32: ON
`-shared`	Instructs the compiler to build a Dynamic Shared Object (DSO) instead of an executable. More...	OFF
`-static`	Enables to link shared libraries (`.so`) statically.	OFF

-syntax	Enables syntax check only. Same as -y. More...	OFF
-Tf*file*	Compiles *file* as a Fortran source. More...	OFF
-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; requires the and support of Streaming SIMD Extensions 2. More...	-tpp6
-u	Sets IMPLICIT NONE by default. 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 uses only n = 0; all other values are NOPs. More...	ON
-uppercase	Changes routine names to all uppercase characters. More...	OFF
-us	Appends (default) an underscore to external subroutine names. More...	ON
-use_asm	Produces objects through the assembler. More...	OFF
-use_msasm IA-32 compiler	Supports Microsoft* style assembly language insertion using MASM style syntax, and if requested, outputs assembly in MASM format. More...	OFF
-V*string*	Displays compiler version information.	OFF

	More...	
-v	Shows driver tool commands and executes tools. More...	OFF
-Vaxlib	Enables linking to portability library (libPEPCF90.a) in the compilation. More...	OFF
-vec- IA-32 compiler	Turn off the vectorizer. 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 integer loops n = 2: indicate vectorized /non-vectorized integer loops n = 3: indicate vectorized /non-vectorized integer loops and prohibit data dependence information n = 4: indicate non-vectorized loops n = 5: indicate non-vectorized loops and prohibit data dependence information More...	-vec _report1
-vms	Enables support for extensions to Fortran that were introduced by Digital VMS and Compaq Fortran compilers. More...	OFF
-w	Suppresses all warning messages. More...	OFF
-w{n}	Suppresses or displays all warning messages. n=0: suppresses all warnings n=1: displays all warnings (default). More...	-w1
-WB	Issues a warning about out-of-bounds array references at compile time. More...	OFF
-wp_ipo	A whole program assertion flag for multifile optimization with the assumption that all user variables and user functions seen in the compiled sources are referenced	OFF

	only within those sources. The user must guarantee that this assumption is safe. More...	
`-x{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. 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. 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`

Functional Group Listings

Compiler Options by Functional Groups Overview

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 groupped 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(TM) 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

Fortran Compilation Environment

See Customizing Compilation Environment section for more information

Option	Description	Default
`-bd,`*`progname`*	Invokes the binder to generate the list of objects required to construct a complete program, given the name of the main program unit within the file. The list is passed to the linker, `ld(1)`.	OFF
`-cl,`*`file`*	Specifies a program unit catalog list to be searched for modules referenced in the program in USE statements.	OFF
`-ic`	Indicates an independent compilation, that is, the FCE of the Intel Fortran Compiler is not accessed or updated. A MODULE or USE statement in the source will cause the compiler to generate an error.	OFF
`-Qinstall`*`dir`*	Sets root directory of compiler installation, indicated in *dir* to contain all compiler install files and subdirectories.	OFF

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
-A[-]	Removes all predefined macros.	-A
-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 module 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 (default)
	n=2: enable only #directives,
	n=3: enable only CVF conditional compilation directives.
		OFF

-I*dir*	Adds directory *dir* to the include file search path.	OFF
-P	Directs the preprocessor to expand your source module 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. (Same as `-Zp{n}`.)	`-align`
`-noalign`	Disables `-align`.	OFF
`-c`	Compile to object only (`.o`), do not link.	OFF
`-falias`	Enables aliasing in program.	ON
`-fno-alias`	Disables aliasing in program.	OFF
`-ffnalias`	Enables aliasing across functions.	ON
`-fno-fnalias`	Disables aliasing across functions, but enables aliasing across calls.	OFF
`-fp`	Disables using `ebp` as general purpose register in optimizations. Directs to use the `ebp`-based stack frame for all functions.	OFF
`-ftz` Itanium(TM)-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
`-Kpic, -KPIC`	Generate position-independent code.	OFF
`-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
`-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.	`-sox`
`-Tffile`	Compile *`file`* as Fortran source.	OFF
`-use_asm`	Produces objects through the assembler.	OFF
`-use_msasm` IA-32 only	Support Microsoft style assembly language insertion using MASM format style and syntax and if requested, output assembly in MASM format.	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. Default: A-32: `-Zp4`, Itanium Compiler: `-Zp8`.	IA-32: `-Zp4` Itanium Compiler: `-Zp8`

Linking

See detailed Linking section.

Option	Description	Default
`-Bdynamic`	Dynamically links libraries at run time. Compared to static linking, results in smaller executables.	OFF
`-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
`-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. For example, `-lm` indicates to link with the math library.	OFF
`-p, -qp`	Compile and link for function profiling with UNIX prof tool.	OFF
`-posixlib`	Enable linking 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`	Enable linking with portability library.	OFF

Compilation Output

See the Specifying Compilation Output section for more information.

Option	Description	Default
`-o`*`file`*	Produce the executable file name specified in *`file`*; for example, `-o`*`myfile`*. 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.	OFF
`-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.	OFF
`-fnoverbose-asm`	Produces assembly file without compiler comments.	OFF

-S	Produce assembly file named file.asm with optional code or source annotations. Do not link.	OFF
-G0	Writes a listing of the source file to standard output, including any error or warning messages. The errors and warnings are also output to standard error, stderr.	OFF
-G1	Prints a listing of the source file to the standard output without INCLUDE files expanded.	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.	OFF

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`*`string`*	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
-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
-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[-]`	Enables (default) or disables assumption of the program's ANSI conformance. Provides cross-platform compatibility .	`-ansi`
`-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.	`-lowercase`

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

Arguments and Variables

See more deatils 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.	OFF
-common_args	Assumes "by reference" subprogram arguments may have aliases of one another.	OFF

`-implicitnone`	Enables the default `IMPLICIT NONE`.	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 to favor code size. Disables option `-fp`. Disables intrinsics inlining to reduce code size. Itanium(TM) 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`	Optimize for speed, but disable some optimizations that increase code size for a small speed benefit. `-O2` disables option `-fp`.	`-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(TM) compiler	Enables/disables the contraction of floating-point multiply and add/subtract operations into a single operation.	`-IFP_fma`
`-IPF_fp_speculation` *`mode`* Itanium compiler	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 current version *`off`*.	`-IPF_fpc64_speculation` *`fast`*

	off: disables fp speculation.	
-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. (-IPF_flt_eval_method2 is not supported in the current version.)	OFF
-IFP_fltacc[-] Itanium compiler	Enables/disables the compiler to apply optimizations that affect floating-point accuracy. By default, the compiler may apply optimizations that affect floating-point accuracy. -IPF_fltacc- disables such optimizations. -IPF_fltacc- is effective when -mp is on.	-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 compiler	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 compiler	Imroves the floating point division-to-multiplication optimization; may impact speed.	OFF
-rcd IA-32 compiler	Enables efficient float-to-integer conversion.	OFF

Processor Dispatch Support (IA-32 only)

See Processor Dispatch Extensions Support for more information.

Option	Description	Default
-tpp5 IA-32 compiler	Optimizes for the Intel Pentium® processor. Enables best performance for Pentium® processor	OFF
-tpp6 IA-32 compiler	Optimizes for the Intel Pentium Pro, Pentium II, and Pentium III processors. Enables best performance for the above processors.	-tpp6
-tpp7 IA-32 compiler	Optimizes for the Pentium 4 and Xeon(TM) processors. Requires the Red Hat version 7.1 and support of Streaming SIMD Extensions 2. Enables best performance for Pentium 4 processor	OFF
-ax{i|M|K|W}	Generates, on a single binary, code specialized to the	OFF

IA-32 compiler	extensions specified by the codes: `i` Pentium Pro, Pentium II processors `M` Pentium with MMX 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.	
`-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 compiler	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
`-nolib_inline`	Disables inline expansion of intrinsic functions.	OFF
`-wp_ipo`	A whole program assertion flag for IPO enabling assumption that all user variables and functions are referenced only within user sources. The user must guarantee that this assumption is safe.	OFF

Profile-guided Optimizations

See detailed Profile-guided Optimizations section.

Option	Description	Default
`-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 compiler	Enables or disables prefetch insertion (requires `-O3`). Reduces the wait time; optimum use is determined empirically.	`-prefetch`
`-scalar_rep[-]` IA-32 compiler	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
`-parallel`	Enables the auto-parallelizer to generate multithreaded code for loops that can be safely executed in parallel.	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=75.

-par_report {0|1|2|3}	Controls the auto-parallelizer's diagnostic levels: 0 - no information 1 - successfully auto-parallelized loops 2 - successfully and unsccessfully auto-parallelized loops 3 - same as 2 plus additional information about any proven or assumed dependences inhibiting auto-parallelization.	-par_report1
-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

Vectorization (IA-32 only)

See deatiled Vectorization section.

Option	Description	Default
-ax{i|M|K|W} IA-32 compiler	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. **Note:** -axi is not a vectorizer option.	OFF
-x{i|M|K|W} IA-32 compiler	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 compiler	Controls the diagnostic messages from the vectorizer with n indicating as follows: n = 0: no information n = 1: vectorized /non-vectorizerd integer loops n = 2: vectorized /non-vectorized integer loops n = 3: vectorized /non-vectorized integer loops and prohibit data dependence information	-vec_report1

	n = 4: non-vectorized loops n = 5: non-vectorized loops and prohibit data dependence information	
-vec- IA-32 compiler	Turns off the vectorizer.	OFF

Optimization Reports (Itanium(TM) Compiler)

See detailed Optimizer Report Generation.

These options are implemented with Itanium(TM) compiler only.

Option	Description	Default
-opt_report Itanium compiler	Generates optimizations report and directs to stderr unless -opt_report_file is specified.	OFF
-opt_report_file*filename* Itanium compiler	Specifies the *filename* to hold the optimizations report.	OFF
-opt_report_level {*min*|*med*|*max*} Itanium compiler	Specifies the detail level of the optimizations report.	-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.	OFF
-opt_report_help Itanium compiler	Prints to the screen all available phases for -opt_report_phase.	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.	OFF

Windows* to Linux* Mapping

Windows* to Linux* Options Cross-reference

This section provides cross-reference table of the Intel® Fortran Compiler options used on the Widows* 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(TM) architecture
- Options available for both IA-32 and Itanium architecture

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	-OF_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.	Windows: /4L72 Linux: -72
/u	-A-	Removes all predefined macros. Issues a warning if OpenMP does not work correctly.	OFF
/align	-align	Analyzes and reorders memory layout for variables and arrays. (Same as -Zp{n}.)	ON
/align-	-noalign	Disables .-align	OFF
/Qansi[-] IA-32 only	-ansi[-]	Enables (default) or disables assumption of the programs ANSI conformance.	ON
/4{Y|N}a	-auto	Causes all variables to be allocated on the stack, rather than in local static storage.	Windows: /4Na Linux: OFF

		Does not affect variables that appear in an EQUIVALENCE or SAVE statement, or those that are in COMMON. Makes all local variables AUTOMATIC.	
/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	Dynamically links libraries at run time. Compared to static linking, results in smaller executables.	OFF
/Qbd,progname	-bd,name	Enables the Intel® Fortran Compiler binder to generate a list of objects to build a PROGNAME.	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

/Qcl,*file*	-cl,*file*	Specifies a program unit catalog list file in which to search for referenced modules.	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} IA-32 only	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.	OFF
/D*name*[=*{#|text}*]	-D*name*[=*{#|text}*]	Defines a macro name and associates it with the specified value.	OFF
/Qdoubletemps	-doubletemps	Ensures that all intermediate results of floating-point expressions are maintained in at least double precision.	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

/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	None	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
None	-falias	Enables aliasing in program.	ON
None	-fno-alias	Disables aliasing in program.	OFF
None	-ffnalias	Enables aliasing across functions.	ON
None	-fno-fnalias	Disables aliasing across functions, but enables 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.	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	OFF

		with the file extensions `.for`, `.f`, or `.ftn`.	
`/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}`	Runs the Fortran preprocessor (`fpp`) on all Fortran source files (`.f`, `.ftn`, `.for`, and `.f90` files) prior to compilation. `n`=0 disable CVF and # directives, equivalent to no `fpp`. `n`=1 enable CVF conditional compilation and # directives (default) `n`=2 enable only # directives `n`=3 enable only CVF conditional directives	`n`=1
`/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 compiler	`-ftz` Itanium compiler	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
`/G0`	`-G0`	Prints source listing to stdout (typically your terminal screen) with the contents of expanded `INCLUDE` files.	OFF
`/G1`	`-G1`	Prints a source listing to stdout, without contents of expanded	OFF

		INCLUDE files.	
/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.	Windows: /4I4 Linux: -i4
/ic	-ic	Runs independent Fortran compilation without accessing and updating Fortran compilation environment (FCE).	OFF
None	-i_dynamic	Enables to link Intel-provided libraries dynamically.	OFF
/I*dir*	-I*dir*	Specifies an additional directory to search for include 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 compiler	-IPF_fma[-] Itanium compiler	Enables/disables the contraction of floating-point multiply and add/subtract operations into a single operation.	ON
/QIPF_fp_speculation *mode* Itanium compiler	-IPF_fp_speculation *mode* Itanium compiler	Sets the compiler to speculate on fp operations in one of the following modes: *fast*: speculate on fp	/QIPF_fp_speculation *fast*

		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; *off*: disables the fp speculation.	
/QIPF_flt_eval_method0 Itanium compiler	-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.	OFF
/QIPF_fltacc[-] Itanium compiler	-IPF_fltacc[-] Itanium compiler	Enables/disables the compiler to apply optimizations that affect floating-point accuracy. By default, the compiler may apply optimizations that affect floating-point accuracy. -IPF_fltacc- disables such optimizations. -IPF_fltacc- is effective when -mp is on.	ON
/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 compiler	-ivdep_parallel Itanium compiler	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*.	
/Qlowercase	-lowercase	Changes routine names to lowercase characters; Linux: also the external symbol names.	Windows: OFF Linux: ON
/Fmfilename	None	Instructs the linker to produce a map file.	OFF
/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	ON

		functions.	
/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
/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	Controls the OpenMP parallelizers diagnostic levels.	Windows: /Qopenmp_ report1 Linux: -openmp

			_report1
/Qopt_report Itanium compiler	-opt_report Itanium compiler	Generates optimizations report and directs to stderr unless -opt_report_file is specified.	OFF
/Qopt_report_file*filename* Itanium compiler	-opt_report_file*filename* Itanium compiler	Specifies the *filename* to hold the optimizations report.	OFF
/Qopt_report_help Iitanium compiler	-opt_report_help Iitanium compiler	Prints to the screen all available phases for -opt_report_phase.	OFF
/Qopt_report_level {*min*|*med*|*max*} Itanium compiler	-opt_report_level {*min*|*med*|*max*} Itanium compiler	Specifies the detail level of the optimizations report.	-opt_report_level*min*
/Qopt_report_phase*phase* Itanium compiler	-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.	OFF
/Qopt_report_routine*routine_substring* Itanium compiler	-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.	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_report1
/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	Windows: /Qpc64 Linux: -pc64
/4{Y|N}posixlib	-posixlib	Enables/disables (Windows) linking to the POSIX library (libPOSF90.a) in the compilation.	Windows: /4Nposixlib Linux: OFF
/Qprec_div IA-32 only	-prec_div IA-32 only	Improve precision of floating-point divides. Some 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	Enables/disables fast float-to-int conversion.	OFF
/S	-S	Produces an assembly output file with optional code.	OFF
/Qsave	-save	Saves all variables (static allocation). Opposite of -auto.	ON
/Qscalar_rep[-] IA-32 only	-scalar_rep[-] IA-32 only	Enables or disables scalar replacement performed during loop transformations (requires -O3).	OFF

/Qsox[-]	-sox[-]	Enables (default) or disables saving of compiler options and version in the executable. Itanium compiler: accepted for compatibility only.	IA-32: ON Itanium compiler: 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
/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.	Windows: /G6 Linux: -tpp6
/4{Y|N}d	-u	Sets IMPLICIT NONE by default.	Windows: /4Yd Linux: 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
None	-use_msasm IA-32 only	Support Microsoft style assembly language insertion using MASM style syntax and if requested, output assembly in MASM format.	OFF
/V*string*	-V*string*	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[-] IA-32 only	-vec[-] IA-32 only	Enables/disables vectorizer.	ON
/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 integer loops n = 2: same as n = 1 plus non-vectorizer integer loops n = 3: same as n = 1 plus dependence information. n = 4: indicate non-vectorized loops n = 5: indicate non-vectorized loops and prohibiting data dependence information.	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

/w90	None	Suppresses warning messages about non-standard Fortran features used.	ON
/w0	-w0	Disables display of warnings.	OFF
/w1	-w1	Displays warnings.	ON
/WB	-WB	Issues a warning about out-of-bounds array references at compile time.	OFF
/Qwp_ipo	-wp_ipo	A whole program assertion flag for multifile optimization with the assumption that all user variables and user functions seen in the compiled sources are referenced only within those sources. The user must guarantee that this assumption is safe.	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(TM) Compiler for Itanium(TM)-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

Note

The Itanium-based applications will not run on an IA-32 system even if they have been developed and compiled with the Itanium cross compiler. See Running Itanium-based Applications Compiled on IA-32 Systems.

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(TM)-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/compiler60/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/compiler60/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(TM)-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 . . .] [linker_options]`
- **Targeted for Itanium architecture:** `prompt>efc [options] file1.f [file2.f . . . .] [linker_options]`

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.
linker_options	`-Ldir` - instruct linker to search `dir` for libraries `-lname` - link with library named *name*

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(TM)-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(TM)** assembly file	Passed to the Intel Itanium assembler.
filename.o	Compiled object module	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

Default Behavior Overview

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
- Searches for any library files in directories specified by the `LIB` variable, if they are not

found 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 Deafult Behavior of the Compiler Options section.
- For IA-32 applications, the compiler uses use `-tpp6` option to optimize the code for the Pentium Pro®, Pentium® II, and Pentium III processors.

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`	`-ansi[-]` 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 setings: specifies the size of real numbers to 8 (IA-32 systems, same as `-autodouble`) or 16 bytes for Itanium(TM) 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.
-sfalign8	Aligns stack for functions with 8 or 16 byte variables. -sfalign16- with 16 byte variables; -sfalign - all functions; -sfalign- - disables stack alignment for all functions.
-u	Sets IMPLICIT NONE.
-us	Appends an underscore to external subroutine names.
IA-32: /Zp4 Itanium systems: /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	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(TM) 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.
-IPF_fltacc Itanium compiler	Enables the compiler to apply optimizations that affect floating-point accuracy. By default, the compiler may apply optimizations that affect floating-point accuracy. -IPF_fltacc- disables such optimizations. -IPF_fltacc- is effective when -mp is on.
-ipo_obj Itanium compiler	Forces the generation of real object files. Requires -ipo. IA-32 systems: OFF
-O, -O1, -O2	Optimize for maximum speed.
-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.

`-tpp6` IA-32 only	Optimizes code for the Pentium Pro®, Pentium II, and Pentium III processors 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`	Enables aliasing in program.
`-ffnalias`	Enables aliasing across functions.
`-fverbose-asm`	Produces assembly file with compiler comments including compiler version and options used.
`-fpp1`	Enables CVF conditional and # directives. `-fpp{0|1|2|3}` runs the Fortran preprocessor (fpp) on Fortran source files (`.f`, `.ftn`, `.for`, and `.f90` files) prior to compilation.
`-sox`	Enables 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-`.
`-w1`	Displays warnings.

Disabling Default Options

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

- Generally, to disable one, group or all 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 option in the command line in this format: `-option-`.
- To disable options that have `{n}` parameter, use `n`=0 version in this format: `-option0`.

Resetting Default Data Types

To reset data type default options, you need to indicate a new option whch 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(TM) 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(TM) Assembler	Intel® Itanium(TM) 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(TM) 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 assembly code file, `file.s`.

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

2. To assemble the file.s file, call Itanium(TM) 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

`-ofile` indicates the output object file name

The above command generates an object file which you can link with the Fortan 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 to 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

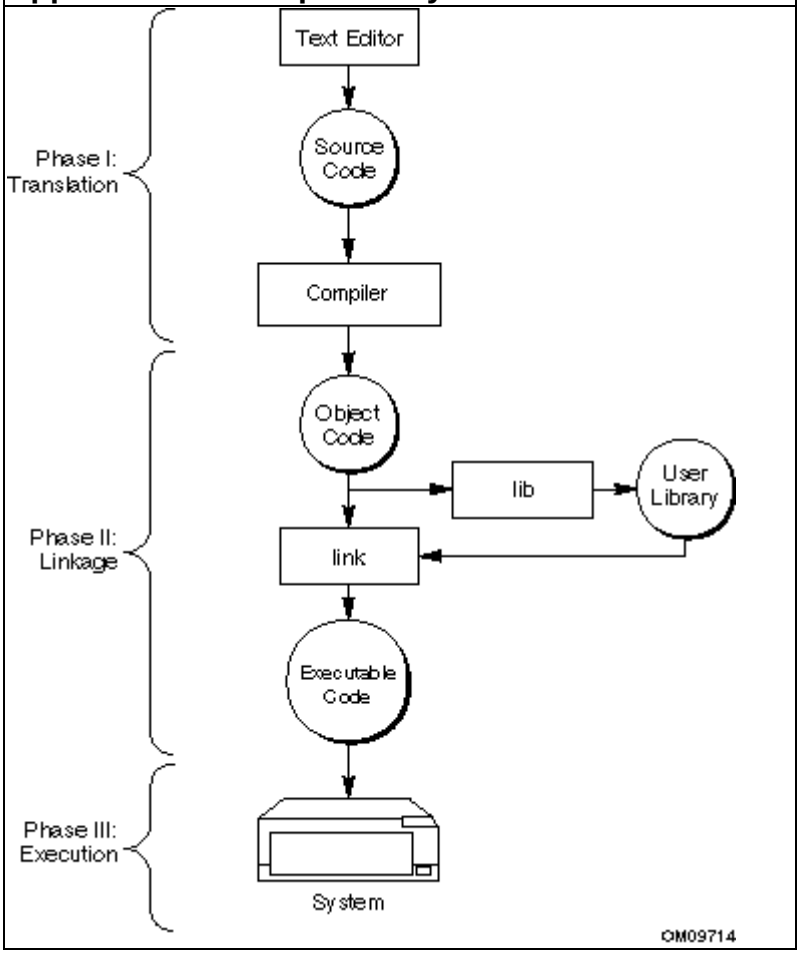
The compiler passes object files and any unrecognized filename to the linker. The linker then determines whether the file is an object file (`.o`) or a library (`.a`). The compiler driver handles all types of input files correctly, thus it can be used to invoke any phase of compilation.

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 modules. 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



Customizing Compilation Environment

Customizing Compilation Environment Overview

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
- FCE options to use FCE tools; for details on FCE structure, see Fortran Compilation Environment (FCE).
- 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

Use the `LIB` and `PATH` environment variables that enable 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.

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 a configuration file, which the compiler should use instead of the default configuration file for the Itanium(TM) compiler.
`IFCCFG`	Specifies a configuration file, which the compiler should use instead of the default configuration file for the IA-32 compiler.
`LIB`	Specifies the directory path for the math libraries.
`F_UFMTENDIAN`	Specifies the numbers of the units to be used for little-endian-to-big-endian conversion purposes.
`INCLUDE`	Specifies the directory path for the include files.
`PATH`	Specifies the directory path for the compiler executable files.
`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 a configuration file, which 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 configuration file in a different location, they can use IFCCFG or EFCCFG environment variable and assign the directory and filename of the .cfg file that needs to be picked up by the compiler.

FCE Options

The following table shows the Fortran Compilation Environment (FCE) options and what you can do with them.

-bd,*progname*	Invokes the binder to generate the list of objects required to construct a complete program, given the name of the main program unit within the file. The list is passed to the linker, ld(1).
-cl,*file*	Specifies a program unit catalog list to be searched for modules referenced in the program in USE statements
-ic	Indicates an independent compilation, that is, the FCE of the Intel Fortran Compiler is not accessed or updated. A MODULE or USE statement in the source will cause the compiler to generate an error.
-Qinstall,*dir*	Sets root directory of compiler installation. The directory indicated in *dir* will contain all compiler install files and subdirectories.

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(TM)-based applications: `efc.cfg`

```
## Sample efc.cfg file for Itanium(TM)-
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(TM)-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. The standard include files are defined in the directories specified in the INCLUDE environment variable. In addition, you can define a specific location of include files with the compiler options, `-Idir` and `-X`. See Searching for Include Files in Preprocessing.

Fortran Compilation Environment (FCE)

FCE Overview

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.

This section describes the essential components of the Intel® Fortran Compilation Environment (FCE) of the Intel Fortran Compiler:

- Object files
- Dictionary files
- Program Unit Catalog Files and Program Unit Catalog List Files
- The FCE Manager Utility
- Binder

The Binder program scans the FCE to create a list of objects required to build the program.

In addition, this section describes the essential structure of Fortran program units and how to compile them: Fortran programs with and without modules and stale program units.

Object Files and Dictionary Files

The Intel Fortran compiler generates one of two file types from your source:

File	Description
Object File (`file.o`)	Compiled from your source by the compiler; the linker uses these files to produce the executable file; generated if the source contains executable code, or if it is a `BLOCK DATA` subprogram.
Dictionary File (`file.d`)	Generated by the compiler if the source contains one or more modules; provides an encoded dictionary of public objects; includes encoding for inter-module object usage.

Program Unit Catalog Files

Program Unit Catalogs are created by the compiler to store the FCE for the executable. Each execution of the Intel® Fortran Compiler command generates critical FCE information, primarily the module information for Fortran 95 programs, and places it in the program unit catalog file (PUCF) `work.pc` in the current compilation directory. This file contains long-lived information and should not be deleted unless it is planned to recompile the entire application from scratch.

The compiler adds the PUCF filename to the list contained in a program unit catalog list file (PUCLF). The default PUCLF file in the installation `/bin` directory is: `/opt/intel/compiler60/ia32/bin/ifc.pcl` or `/opt/intel/compiler60/ia64/bin/efc.pcl`.

At installation, you will see the following entries in this file:

IA-32 compiler:

`work.pc:` the PUCF in the user's current directory
`<installation directory>/bin/ifc.pcl:` the PUCLF in the installation directory

Itanium(TM) compiler:

`work.pc` : the PUCF in the user's current directory
`<installation directory>/bin/efc.pcl`: the PUCLF in the installation directory

Specifying the Name and Path of the PUCLF

The default PUCLF is shared by all users of the compiler installation. Therefore, you may prefer to specify a different name for the PUCLF file with `-cl`. For example, to compile `file.f` in the current directory, type the following:

IA-32 compiler:

```
ifc -cl,myfile.pcl file.f
```

Itanium compiler:

```
efc -cl,myfile.pcl file.f
```

This will add to the default or create a PUCLF `myfile.pcl` in the current directory. You may add entries for additional PUCF files with a text editor, or by specifying in the `-cl` parameter of a subsequent compilation another PUCLF (including the path) with the entries you need.

The order of program unit catalogs within a program unit catalog list file determines the order in which the compiler searches for catalogs during compilation. You can share FCEs among modules with non-concurrent compilations. For example, if two catalogs contain the module referenced in the USE statement, the compiler selects the first version referenced. However, within a single catalog, the names of program units must be unique. Violating this restriction can cause some of your programs to be built incorrectly.

You can specify the file path for external modules in a program unit catalog list file. You can create or modify this file with any text editor to give access to the modules referenced in the USE statements.

Guidelines for the PUCLF

Observe these guidelines when creating or editing a program unit catalog list file:

- In the first line, specify the file name of the work catalog.
- In succeeding lines, you can specify the full path names of other program unit catalogs in which to search.

By default the compiler creates a catalog list file named `ifc.pcl` or `efc.pcl` with the following entry in it: `work.pc`. The default PUCLF name can be changed with the `-cl` option parameter either on the command line or in the configuration file `ifc.cfg` or `efc.cfg`.

To use modules compiled in other directories, you can explicitly create your own program unit catalog list file and use whatever file name you want; for example, `mywork.pcl`.

Your catalog list file `mywork.pcl` might contain the following:

```
work.pc
 /home/usr/mod1/work.pc
 /home/usr/mod2/work.pc
```

Note:
Make sure to never use blanks in the directory names.

An Example of Development Organization

Consider a project involving a number of developers, each requiring the capability to build a test version of the software. The project consists of a mix of common program units and other program units trusted to work correctly and used by individual programmers. A suitable organization might be as follows:

- Trusted common program units are compiled in a number of directories:

  ```
  /usr/trusted1, /usr/trusted2, ... , /usr/trustedn.
  ```

- Each user specifies a directory in which program units are compiled. Each directory contains a program unit catalog list file with the contents as follows:

  ```
  myownwork.pc
   /home/trusted1/trusted.pc
   /home/trusted2/trusted.pc
         :
         :
   /home/trustedn/trusted.pc
  ```

where `myownwork.pc` is a developer's personal work catalog, and the trusted common program units are referenced by the `trusted.pc` program unit catalogs in their respective directories.

Since each developer has a private work catalog, concurrent compilations cannot interfere with each other. Further, shared concurrent compiler access to the trusted common program units is easier.

The FCE Manager Utility

The FCE Manager (FCEM) is a utility that enables you to interrogate and update program unit catalogs belonging to an FCE. It is activated by the command `ifccem` (IA-32 compiler) or `efccem` (Itanium(TM) compiler) and by default prompts for commands from the keyboard. However it may also be operated in script files as follows:

IA-32 compiler:

```
ifccem <<!
 commands
 !
```

Itanium compiler:

```
efccem <<!
 commands
 !
```

To obtain information on the set of commands available, use the command h (help). If h (help) is followed by the name of a command, it provides a detailed explanation of that command. The command q (quit) terminates execution of the FCEM.

Note :

When you are developing your Itanium-based application, and the application contains MODULEs, you must be careful to compile all of your code on the same host, regardless of the target platform. For example, if you are developing applications for an Itanium-based platform on an IA-32 host, you must compile all of your code on the IA-32 host. You cannot use a work.pc (program catalog) file generated on one platform when compiling on another platform. Also, you must use the FCE tool for the host where you compiled your code, rather than the FCE tool for the other platform.

The table that follows lists FCE manager commands with brief descriptions.

FCE Manager Commands

Command	Description	Syntax
cl	Clear a program unit catalog.	cl <puc>
	Example: cl test.pc Removes all program units from program unit catalog test.pc.	
co	List compilation order	co pu puclist
	Examples: co LIST test.pc Lists a valid compilation order for program units belonging to the program LIST, and sought in program unit catalog test.pc. co MAIN.PROGRAM <puclist> Lists a valid compilation order for program units belonging to MAIN.PROGRAM, and sought in the program unit catalogs whose names are given by puclist.	
cp	Copy program units	cp from_puc to_puc pulist
	Examples: cp test.pc test2.pc Copies all program units from test.pc to test2.pc. cp test.pc test2.pc A B Copies program units A and B from program unit catalog test.pc to program unit catalog test2.pc.	
cr	Create a program unit catalog	cr puc
	Example: cr test.pc Creates the new program unit catalog test.pc.	
fi	Find a program unit	fi pu puclist

	Examples:
	`fi EX test.pc test2.pc`
	Finds program unit `EX` in program unit catalogs `test.pc` and `test2.pc`.
	`fi TEST <puclist>`
	Finds program unit `TEST` in the program unit catalogs in file puclist.	

`fu`	Find users of a program unit	`fu pu puclist`

	Examples:
	`fu MOD test.pc test2.pc`
	Finds users of module `MOD` in progr.unit catalogs `test.pc` and `test2.pc`.
	`fu MOD2 <puclist>`
	Finds users of `MOD2` in the program unit catalogs specified in file `puclist`.	

`h`	Provide help information	`h [command]`

	Examples:
	`h`
	Lists all the available FCEM commands.
	`h rm`
	Lists help information about the command `rm`.	

`ls`	List program units	`ls [options] puc [pulist]`

	Examples:
	`ls test.pc`
	Produces a brief listing of program units in program unit catalog `test.pc`.
	`ls /al test.pc`
	Produces a full listing of program units in `test.pc` in alphabetic order.
	`ls /l /t test.pc B Z C A`
	Produces a full listing of program units `B`, `Z`, `C` and `A` in program unit catalog `test.pc`, in order of creation date/time.	

`mo`	Modify recorded object file names	`mo name puc [boldest]`

	Examples:
	`mo mylib.a test.pc`
	Modifies all recorded object file names of program units in the program unit catalog `test.pc` to indicate their presence in the object library `mylib.a`.
	`mo newobj.o work.pc obj.o`
	In the program unit catalog `work.pc`, modifies those program units which have a recorded object file name of `obj.o` to have the recorded object file name `newobj.o`.
	`mo mylib.a test.pc obj1.o`	

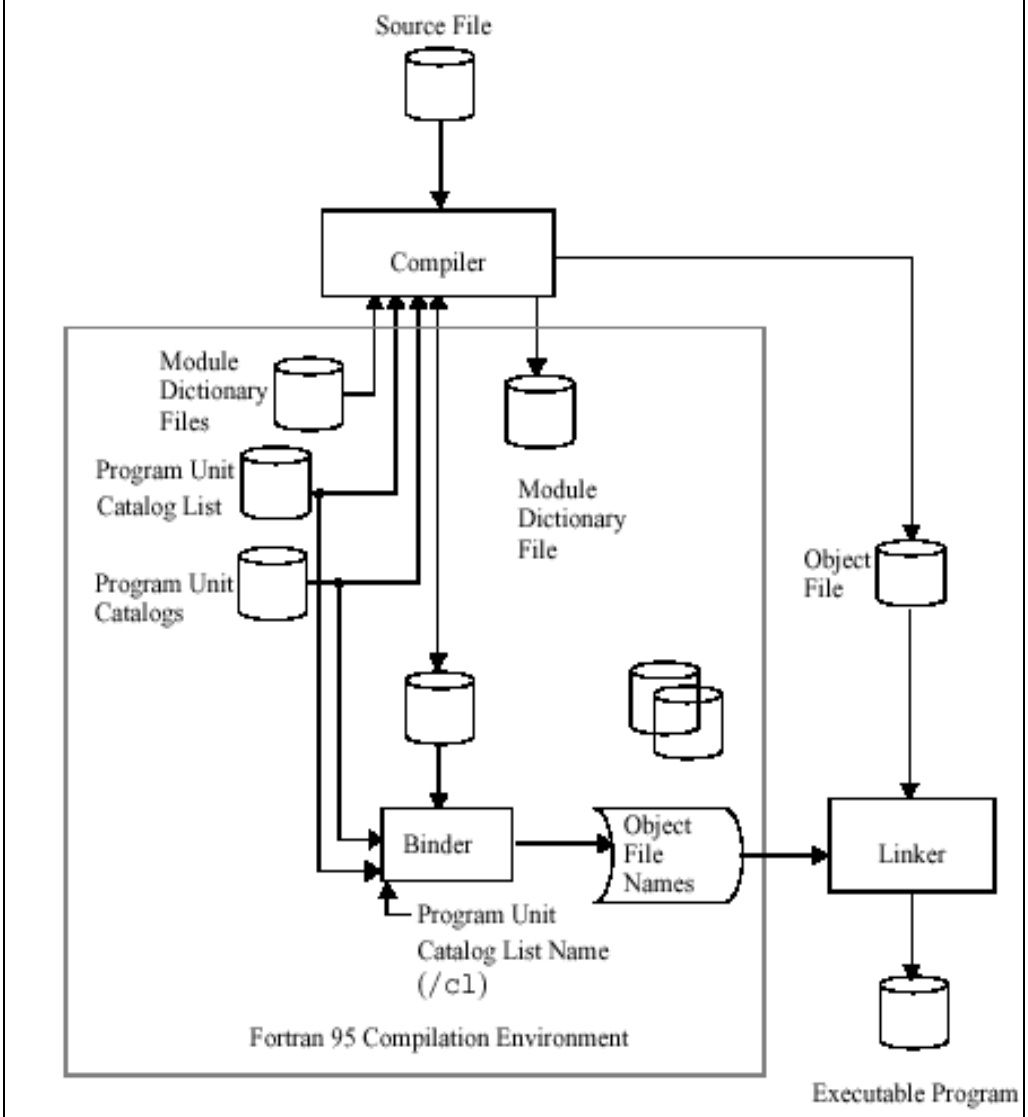
	`oldlib[obj2.o]` In the program unit catalog `test.pc`, modifies those program units which have a recorded object file name of `obj1.o` or `oldlib[obj2.o]` so that the recorded object file name indicates its presence in library `mylib.a`.	
`q`	Terminates execution of `ifccem`.	`q`
	Example: `q`	
`rm`	Remove program units	`puc pulist`

	Examples:
	`rm test.pc A`
	Removes program unit A from program unit catalog `test.pc`.
	`rm test2.pc A B C`
	Removes program units A, B, and C from program unit catalog `test2.pc`.	

The Binder

The binder is a program activated by the compiler option `-bd`, which scans an FCE to generate the list of objects required to build the program. It then presents the list to `ld(1)` for linking. The figure below shows how the binder relates to the rest of the FCE.

Intel Fortran Compilation Environment with the Binder



Activating the Binder

The format of the option -bd is the following:

`-bd,`*`mainprogramname`*

where `mainprogramname` is the name specified in the `PROGRAM` statement of the main program, or is `MAIN.PROGRAM` if no `PROGRAM` statement is present.

A command line invoking `ifc` (IA-32 compiler) or `efc` (Itanium(TM) compiler) to compile Fortran source can also include a `-bd` option to invoke the binder; in this case, the results of the compilation are available to the binder.

The binder assumes that all objects belonging to the program are in the FCE defined by the program unit catalog list file specified by option `-cl` or by `work.pc` if the option `-cl` is not specified. Any other objects, for example non-Fortran objects, that are required in the linking stage, must be specified explicitly through the compiler.

Advantages of Using the Binder

The binder provides three principal advantages:

- It automatically defines the objects to be included in a large scale project.
- It detects and flags stale modules, as described in the preceding section.
- It searches program unit catalogs in the order specified in the program unit catalog list file, so enabling the user to distinguish between identically named program units in different catalogs.

The use of the binder is not mandatory. Objects may be specified explicitly on the compiler command invocation line if desired.

Dependent and Independent Compilation

You can independently compile units that comprise a Fortran program. These units include the following:

- main program
- external subroutines
- external functions
- block data subprograms

Prior to Fortran 90, compilation of a program unit did not require data from the compilation of another unit. Consequently, the order of compilation of units did not affect the output.

For Fortran 95 programs, this is not always the case. The addition of modules to the language introduces a compilation dependence. A module can reference other program units with the USE statement. In contrast to independent units, dependent units require data from another module that must be compiled first. Thus, the dependence introduces an order that you must follow to compile program units.

You can compile the dependent and independent units in the same source module or in separate source files. However, the dependent file must compile after the file on which it depends.

Fortran Programs with or without Modules

There are two ways of working with multimodule programs depending on the scale of your project.

Small-Scale Projects

In a small-scale project, the source files are in a single directory, so module management is not an issue. A simple way to compile and use modules is to incorporate a module before a program unit that references it with USE. In this case, sources may be compiled and linked in the same way as FORTRAN 77 sources; for example if `file1.f` contains one or more modules and `file2.f` contains one or more program units that call these modules with the USE directive. The sources may be compiled and linked by the commands:

IA-32 applications:

```
ifc file1.f file2.f
```

or

`ifc -c file1.f` (where `-c` option stops the compilation after an `.o` file has been created)

```
ifc file1.o file2.f
```

Itanium(TM)-based applications:

```
efc file1.f file2.f
```

or

efc -c *file1.f* (where -c option stops the compilation after an .o file has been created)

efc *file1.o file2.f*

Larger-Scale Projects

In a larger-scale software project, module management becomes a significant issue. The Intel Fortran Compiler incorporates the following features to ease this task:

variable grouping of program units in program unit catalogs

- variable module search path
- detection of stale program units
- utilities to find, copy, delete and display program unit catalog entries
- program binder to construct an inventory of objects for linking

By default, ifc (IA-32 compiler) or efc (Itanium compiler) compiles each program unit for multimodule usage in the FCE. If you wish to specify independent compilation, use the -ic option:

IA-32 compiler:

ifc -ic *file.f*

Itanium compiler:

efc -ic *file.f*

Note

The current version of the Intel Fortran Compiler does not support STRUCTURES or Cray pointers within the Fortran modules.

Fortran Programs Without Modules

If you do not use modules in your programs, you can still benefit from the FCE through the use of its binder. The binder provides features to automate your compilation tasks and expedite your application development. These features are part of the FCE structure.

Stale Program Units

When a program unit, M1, uses a module, M2, the compilation of M1 is up-to-date if it occurred after the latest compilation of M2. Otherwise, module M1 is stale and may require recompilation. Stale program units often occur in large-scale development. They are detected and flagged both by the compiler and by the binder. A typical scenario involves at least three sources, file1.f, file2.f and file3.f, and a compilation sequence as shown in the following example.

Example of Compilation Sequence without a Stale Program Flag

```
file1.f
 module mod1
  :
 end module mod1
 file2.f
 module mod2
  :
  :
```

```
 use mod1
  :
 end module mod2
 file3.f
 program p
  :
 use mod2
  :
end program p
```

The table that follows shows the compilation sequence for IA-32 applications without and with issuing the stale program flag. The same sequence is used for Itanium-based applications with the `efc` command instead of `ifc`. The left column reflects a small-scale project with the program files compiled in proper order. The right column reflects possibly a larger-scale program compilation. Program `P` had been compiled with the binder option, `-bd`, right after `file1.f` had been edited and recompiled, while `file2.f` (which uses `mod1` from `file1.f`) had not been recompiled. In such a case, the binder flags the module `mod2` as stale and issues a message. The programmer then has to recompile `mod2`.

Stale Program Flag	
No Stale Program Flag	**Stale Program Flag Issued**
edit file1.f *etc*	ifc -c file1.f
ifc -c file1.f	ifc -c file2.f
ifc -c file2.f	ifc -c file3.f
ifc file3.f	*edit* file1.f
	ifc -c file1.f
	ifc -bd, P 3

Customizing Compilation Process

Customizing Compilation Process Overview

This section describes options that customize compilation processpreprocessing, 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 libraries used by compiler to which you can link, compiler diagnostics, and mixing C and Fortran in the respective 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(TM) 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 a *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(TM)-based applications:

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

Preprocessing

Preprocessing Overview

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
```

Itanium(TM)-based applications:

```
prompt>efc source.fpp
```

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 Compiler Options

You can enable 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`, `.for` or `.f90` suffix in the mode set by `n`:

`n`=0: disable CVF and `#directives`; equivalent to no fpp

`n` =1: enable CVF conditional compilation and `#directives` (`-fpp1` is default)

`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. The `cpp` prohibits to use the string constant value in `#if` expression. So `fpp` won't support it either.

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 module 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(TM)-based applications:

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

When you specify the `-P` option, the preprocessor expands your source module 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 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. 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(TM)-based applications or in command line.

Specifying an Include Directory, `-I`

Included files are brought into the program with a `#include` preprocessor directive or a Fortran `INCLUDE` statement. Use the `-Idir` 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

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(TM)-based applications:

```
prompt>efc -X -I/alt/include newmain.f
```

Defining Macros, `-D`, `-U` and `-A`

You can use the `-D` option to define the assertion and macro names to be used during preprocessing. The `-U` option directs the preprocessor to suppress an automatic definition of a macro.

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 `-Dname[=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 in quotation marks if it contains non-alphanumerics.

Preprocessing replaces every occurrence of *name* with the specified *value*. For example, to define a macro called *SIZE* with the *value100* use the following command:

IA-32 applications:

```
prompt>ifc -DSIZE=100 prog1.f
```

Itanium(TM)-based applications:

```
prompt>efc -DSIZE=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*.

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.

To remove all of the predefined macros, use the -A option. Note that the -A- option issues a warning if OpenMP function does not work correctly.

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	Disable	Description - When Used
IA-32 and Itanium compilers			
_M_IX86=*n*	ON,*n*=600	_u	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
IA-32			
__linux__	ON	_u	Defined for Linux applications
__IFC	ON	no	Identifies the Intel Fortran Compiler
Itanium compiler			
_M_IA64_linux	ON	_u	Defined for Itanium-based Linux applications
__EFC	ON	no	Identifies the Intel Fortran Compiler

Compiling

Compilation Overview

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.

Compilation Options

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 -lm
```

Itanium™ compiler:

```
prompt>efc a.f -lm
```

Aliasing

The following options enable or disable compiler aliasing capability:

- `-falias` enables aliasing within a program
- `-fno-alias` disables aliasing within a program
- `-ffnalias` enables aliasing across funcrions
- `-fno-fnalias` disables aliasing across funcrions, but enables aliasing across calls

Translating Other Code to Fortran

With the `-Tf`*`file`* option, you can compile some other than Fortran code file as Fortran (translate to Fortran).

For example:

```
prompt>ifc -Tfa.c b.f
```

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

Saving Compiler Version and Options Information, `-sox`

You save the compiler version and options information in the executable with `-sox`. The `-sox` option is enabled by default, which 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.

The size of the executable on disk is increased slightly by the inclusion of these information strings. If this is a concern, you can specify `-sox-` to disable this feature.

Note that for Itanium(TM)-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(TM)-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.

Monitoring Data for IA-32 Systems

Correcting Computations for IA-32 Processors, -0f_check

Specify the -0f_check option to avoid the incorrect decoding of the instructions, which 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

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 Debugging and Optimizations. The -fp option is disabled by default or when -O1 or -O2 (see optimization-level options) are specified.

Monitoring Data for Itanium(TM)-based Systems

Flushing to Zero Denormal Values, -ftz

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 to 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 the 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 base data types 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, which use these unit numbers, will perform relevant conversions. Other READ/WRITE statements will work in the usual way.

In general case, the variable consists of two parts divided by 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 diapason of units. Number of list members is limited by 64. decimal is a non-negative decimal number less than 2^{32}.

Converted data should have base 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 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.

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
 C     on 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

Specifying Compilation Output Overview

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 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 `-ofile` 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 `-ofile` 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 `-ooutfile` option, the command above produces an executable file named `a.out`, the default iexecutable 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 `-ofile` options to specify an alternate name for an object file.

For example:

IA-32 compiler:

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

Itanium(TM) 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 anme, 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 `–ofile` 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 `-ofile.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.

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`	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(TM) 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.
See Assembly File Code Example.

Assembly File Code Example

The following is an example of a portion of an assembly file code for IA-32:

```
main:
 .B1.1:
 /1
 / Preds .B1.0
 / AFL
 pushl     %ebp                /1.0 1 0
 movl      %esp, %ebp          /1.0 1 0
 subl      $3, %esp            /1.0 1 0
 andl      $-8, %esp           /1.0 1 0
 addl      $4, %esp            /1.0 1 0
 subl      $12, %esp           /1.0 1 0
 movl      12(%ebp), %edx      /1.0 2 0
 leal      8(%ebp), %eax       /1.0 4 0
 movl      %eax, (%esp)        /1.0 4 0
 movl      %edx, 4(%esp)       /1.0 5 0
 call      f90_init            /1.0 6 0
 / LOE ebx esi edi
```

The elements in the above code are as follows:

- `.B1.1`: identifies the beginning of the first basic block in the first function of the file. A basic block is a set of instructions with the property that if the first instruction is executed then all of the subsequent instructions in the set are also executed.

- /1 following the basic block label is the block execution count. This count is only printed when the -prof_use option is used. It indicates the average number of times a block was executed when the instrumented program was run. See Profile-Guided Optimization for more information on -prof_use.
- /Preds is a list of predecessors of the current basic block. Predecessors are blocks that can transfer control to the current basic block.
- The numbers (1.0) following the slash (/) at the end of each instruction indicate the source line number and column corresponding to that assembly language instruction.
- /LOE indicates a list of registers which are live on exit from the current basic block. These are registers that contain values to be used by succeeding basic blocks.

Itanium(TM)-based applications:

An assembly file code portion:

```
        .section .text
 // -- Begin  main
 .proc  main#
 .align 32
 // Block 0: entry  Pred:     Succ: 3
 // Freq 1.0e+000, Prob 1.00, Ipc 2.67
 .global main#
 .align 32
 main:
 {   .mmi
 alloc   r34=ar.pfs,2,2,2,0      //0:  1
 add     sp=-64,sp               //0:  1
 nop.i   0 ;;
 } { .mii
 ld8     r30=[sp]                //1:  1
 mov     r35=b0                  //1:  1 MS
 add     r36=$2$1_2pab_p$0# + _2$1_2auto_size -
0x00000030,sp
                                 //1:  1 MS
 RE
 } { .mmi
 mov     r37=r33 ;;              //1:  1 MS
 st8     [r36]=r32               //2:  1 B3 DS
 mov     r32=gp                  //2:  1 B3 DS
 // Block 3:  Pred: 0     Succ: 1
 // Freq 1.0e+000, Prob 1.00, Ipc 1.00
 } { .mib
 nop.m   0
 nop.i   0
 br.call.sptk.many       b0=f90_init# ;;    //2:  1 MS
```

The elements in the above code are as follows:

- ; Prob 1.00 indicates the probability assigned to a jump.
- Each curly brace pair { } indicates an instruction bundle. A bundle is a group of up to three instructions that may execute simultaneously if there are no stalls or dependencies.
- main is a label that starts the program
- // indicate comments
- [ ] indicate indirect addressing

For more information, see *Intel® Itanium™ Assembler User's Guide*.

Listing Options

The following options produce messages to the standard output, which by default is the screen.

- The `-G0` option writes a listing of the source file to standard output, including any error or warning messages. The errors and warnings are also output to standard error, `stderr`.
- The `-G1` option prints a listing of the source file to the standard output without `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`	Dynamically links libraries at run time. Compared to static linking, results in smaller executables.
`-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, `-lm` indicates to link with the math library.
`-L`*`dir`*	Instructs linker to search *`dir`* for libraries.
`-posixlib`	Enable 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.
`-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(TM) compiler:

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

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

Debugging

Debugging Options Overview

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:

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

Preparing for 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(TM)-based applications: `prompt>efc -g prog1.f`

Debugging and Assembling

The compiler supports the generation of debugging information in assembly files. If you specify the `-g` option with `-S`, the assembly file will contain the debugging ionformation. If you further produce an object file, the resulting object file will contain debugging information. If you link the object file and then use the GDB debugger on it, you will get full symbolic representation.

Support for Symbolic Debugging

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.

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(TM)-based applications:

```
prompt>efc -y prog1.f
```

Compiling Source Lines with Debugging Statements, -DD

Use the -DD option to compile source lines containing user debugging statements. Debugging statements included in a program are indicated by the letter D in column 1. By default, the compiler takes no action on these statements. 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.

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 was 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 Conformance Options

Fortran Language Options Overview

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

TThe `-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[-]` 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.

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`, 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.The default (with -O2 ON) corresponds to -auto_scalar-. Opposite of -auto. To disable -save, set -auto.

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 DOUBLE PRECISION and INTEGER 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 enable the default IMPLICIT NONE.

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 routine to be generated.

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.

Optimizations

Optimization Levels

Optimization Levels Overview

Each of the command-line options: `-0`,`-01`, `-02` and `-03` turn on several compiler capabilities. `-0` and `-01` are practically the same and mentioned both for compatibility with other compilers. See the summary of these options.

The following table summarizes the optimizations that the compiler applies when you invoke `-01` and/or `-02`, or `-03` optimizations.

Option	Optimization	Affected Aspect of Program
`-01, -02`	global register allocation	register use
`-01, -02`	instruction scheduling	instruction reordering
`-01, -02`	register variable detection	register use
`-01, -02`	common subexpression elimination	constants and expression evaluation
`-01, -02`	dead-code elimination	instruction sequencing
`-01, -02`	variable renaming	register use
`-01, -02`	copy propagation	register use
`-01, -02`	constant propagation	constants and expression evaluation
`-01, -02`	strength reduction-induction variable	simplification instruction, selection-sequencing
`-01, -02`	tail recursion elimination	calls, further optimization
`-01, -02`	software pipelining;	calls, further optimization; for Itanium-based applications, `-01` turns off software pipelining to reduce the code size
`-03`	prefetching, scalar replacement, loop transformations	memory access, instruction parallelism, predication, software pipelining

Setting Optimization Levels

For IA-32 and Itanium(TM) architectures, the options can 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
-O1	Optimizes to favor code size. Enables the same optimizations as -O except for loop unrolling and software pipelining. At -O1 the global code scheduler is tuned to favor code size. -O and -O2 turn the software pipelining ON. Generally, -O or -O2 are recommended over -O1.

IA-32 Compiler

Option	Effect
-O, -O1, -O2	Optimize to favor code speed. Disable option -fp. The -O2 option is ON by default. Inlines intrinsics.
-O3	Enables -O2 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 -O2. 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
-O2	ON by default. -O2 turns ON intrinsics inlining. 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.

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 Optimizations

Floating-point Arithmetic Precision Overview

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

The `-mp`, `-mp1`, and `-doubletemps` options are used by both architectures. Use the `-doubletemps` option to ensure that all intermediate results of floating-point expressions are maintained in at least double precision, both for IA-32 and Itanium-based applications.

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 Maintaining and 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

Use `-prec_div` to improve the floating point division-to-multiplication optimization. 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, but is faster.

-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.

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 improve the performance of code that requires floating point-to-integer conversions. The optimization is obtained by controlling the change of the rounding mode.

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(TM)-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 *`modes`*:

`fast`: sets the compiler to speculate on floating-point operations;

`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/disables the compiler to apply optimizations that affect floating-point accuracy. By default, the compiler applies optimizations that affect floating-point accuracy. `-IPF_fltacc-` disables such optimizations. `-IPF_fltacc-` is effective when `-mp` is on.

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

Maintaining and 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.

Processor Dispatch Extensions Support (IA-32 Only)

Targeting a Processor and Extensions Support Overview

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 `-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{5|6|7}`	`-tpp5` Pentium® processor. `-tpp6` Pentium Pro, Pentium II, and Pentium III processors. Default. `-tpp7` Pentium 4 and Xeon(TM) processors. Requires the RedHat version 7.1 and support of Streaming SIMD Extensions 2.
`-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}

For IA-32-targeted compilations, 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...
Pentium® processor and Pentium processor with MMX(TM) technology	-tpp5
Pentium Pro, Pentium II and Pentium III processors	-tpp6 (default option)
Intel® Pentium® 4 and Xeon(TM) processors	-tpp7

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

```
prompt>ifc prog.f

prompt>ifc -tpp6 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 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**	**Pentium 4, Xeon(TM) Processors**
Pentium Processor	-tpp5	-tpp5	-tpp6	-tpp6	-tpp6	-tpp7
Pentium Processor with MMX technology	N-A	-tpp5, -xM	-tpp6	-tpp6, -xM	-tpp6, -xM	-tpp7, -xM
Pentium Pro Processor	N-A	N-A	-tpp6, -xi	-tpp6, -xi	-tpp6, -xi	-tpp7, -xi
Pentium II Processor	N-A	N-A	N-A	-tpp6, -xiM	-tpp6, -xiM	-tpp7, -xiM
Pentium III Processor	N-A	N-A	N-A	N-A	-tpp6,	-tpp7,

					-xK	-xK
Pentium 4, Xeon Processors	N-A	N-A	N-A	N-A	N-A	-tpp7, -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 32-bit processors as well, you would use the following command line:

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

Interprocedural Optimizations (IPO)

IPO Overview

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 Overview

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 and `ifc`, or using the `xild` tool. (See Creating a Multifile IPO Executable Using a Project Makefile.)

> **Note**
>
> Failure to link "mock" objects with `ifc -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.

`-wp_ipo` is a whole program assertion flag that tells the compiler the whole program is present. It enables multifile optimization with the whole program assumption that all user variables and user functions seen in the compiled sources are referenced only within those sources. This is an unsafe option. The user must guarantee that this assumption is safe.

Compilation with Real Object Files, `-ipo_obj`

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 `xild` or `xild -lib`. 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 `xild` or `xild -lib`, 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 `xild`.
- 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 Multifile IPO Executable

The following table explains how to enable multifile IPO for compilations targeted for IA-32 hosts and for compilations targeted for Itanium(TM)-based systems.

IA-32 systems	Itanium(TM)-based systems
Compile your modules with -ipo as follows: 1. `prompt>`**`ifc -ipo -c a.f b.f c.f`** Use `-c` to stop compilation after generating `.o` files. Each object file has the IR for the corresponding source file. With preceding results, you can now optimize interprocedurally: 2. `prompt>`**`ifc -o`*`nu_ipo_file`*` -ipo a.o b.o c.o`** The `-o`*`name`* option stores the executable in *`nu_ipo_file`*. Multifile IPO is applied only to modules that have an IR, otherwise the object file passes to link stage. For efficiency, combine steps 1 and 2: `prompt>`**`ifc -ipo -o`*`nu_ipo_file`*` a.f b.f c.f`**	Compile your modules with -ipo as follows: 1. `prompt>`**`efc -ipo -c a.f b.f c.f`** Use `-c` to stop compilation after generating `.o` files. Each object file has the IR for the corresponding source file. With preceding results, you can now optimize interprocedurally: 2. `prompt>`**`efc -o`*`nu_ipo_file`*` -ipo a.o b.o c.o`** The `-o`*`name`* option stores the executable in *`nu_ipo_file`*. Multifile IPO is applied only to modules that have an IR, otherwise the object file passes to link stage. For efficiency, combine steps 1 and 2: `prompt>`**`efc -ipo -o`*`nu_ipo_file`*` a.f b.f c.f`**

See Using Profile-Guided Optimization: An Example for a description of how to use multifile IPO with profile information for further optimization.

Creating a Multifile IPO Executable Using a Project Makefile

Most applications use a make file or something similar to call a linker such as `ld(1)`. This is done automatically when you compile and link with `ifc`. Therefore, when `-ipo` must result in a separate linking step, you must use the linker driver xild instead, as follows:

`prompt``>`**`xild -ipo <LINK_commandline>`**

where:

`-ipo`	enables additional IPO diagnostic output (optional)
`<LINK_commandline>`	is your linker command line

Use the `xild` syntax when you use a makefile instead of step 2 in the example Creating a Multifile IPO Executable. The following example places the multifile IPO executable in *`filename`*: `prompt>`**`xild -o`*`filename`*` a.o b.o c.o`**

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.

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 in-lining and the minimum in-lining criteria, see Criteria for Inline Function Expansion and Controlling Inline Expansion of User Functions.

Using `-ip` witn `-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).

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=FALSE	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 is set to the maximum number of 200.
-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(TM) compiler**: ip_ninl_min_stats = 15
-ip_ninl_max_total_stats=n	Sets the maximum increase in the total_stats. The number of intermediate language statements for each function that is expanded in line. The number n is a positive integer. By default, each function can increase to a maximum of 5000 statements.
-ip_no_external_ref	Indicates that the source file contains the main program and does not contain functions that are referenced by external functions. If you do not specify this option, the compiler retains an original copy of each expanded in-line function.

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 -Qoptionf,-ip_ninl_max_stats=5 source.f
```

Inline Expansion of Functions

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

  `-Qoptionf,-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 in-line expansions will provide the greatest benefit to program performance. This is done using the following default heuristics:

- The default heuristic focuses on call-sites in loops or calls to functions containing loops.
- When profile information is available, the focus changes to the most frequently executed call-sites.
- Also, the default in-line heuristic does not permit the inlining of functions with more than 230 intermediate statements, or the number specified by the option `-Qoptionf,-ip_inline_max_stats`. The default inline heuristic will stop inlining when direct recursion is detected.

  The default heuristic will always inline very small functions that meet the minimum inline criteria. By default, functions with 7 or fewer intermediate statements will be inlined. This limit can be modified with the option `/Qoption,f,/ip_ninl_min_stats=new value`. See -Qoption Specifiers.

When you use profile-guided optimizations, a number of other heuristics are used (see Profile-Guided Optimization (PGO)).

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.

Profile-guided Optimizations

Profile-guided Optimizations Overview

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.

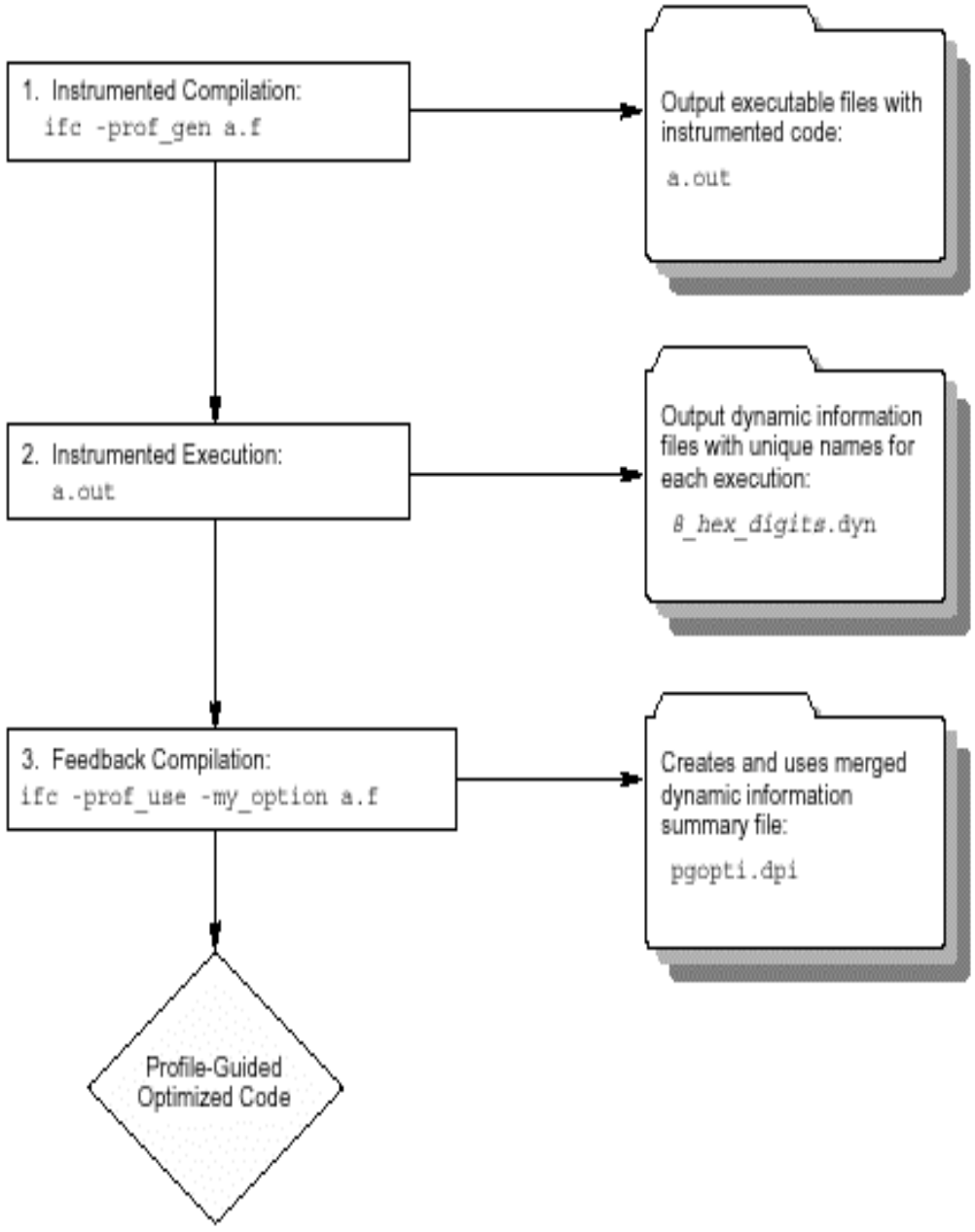
PGO Phases

The PGO methodology requires three phases:

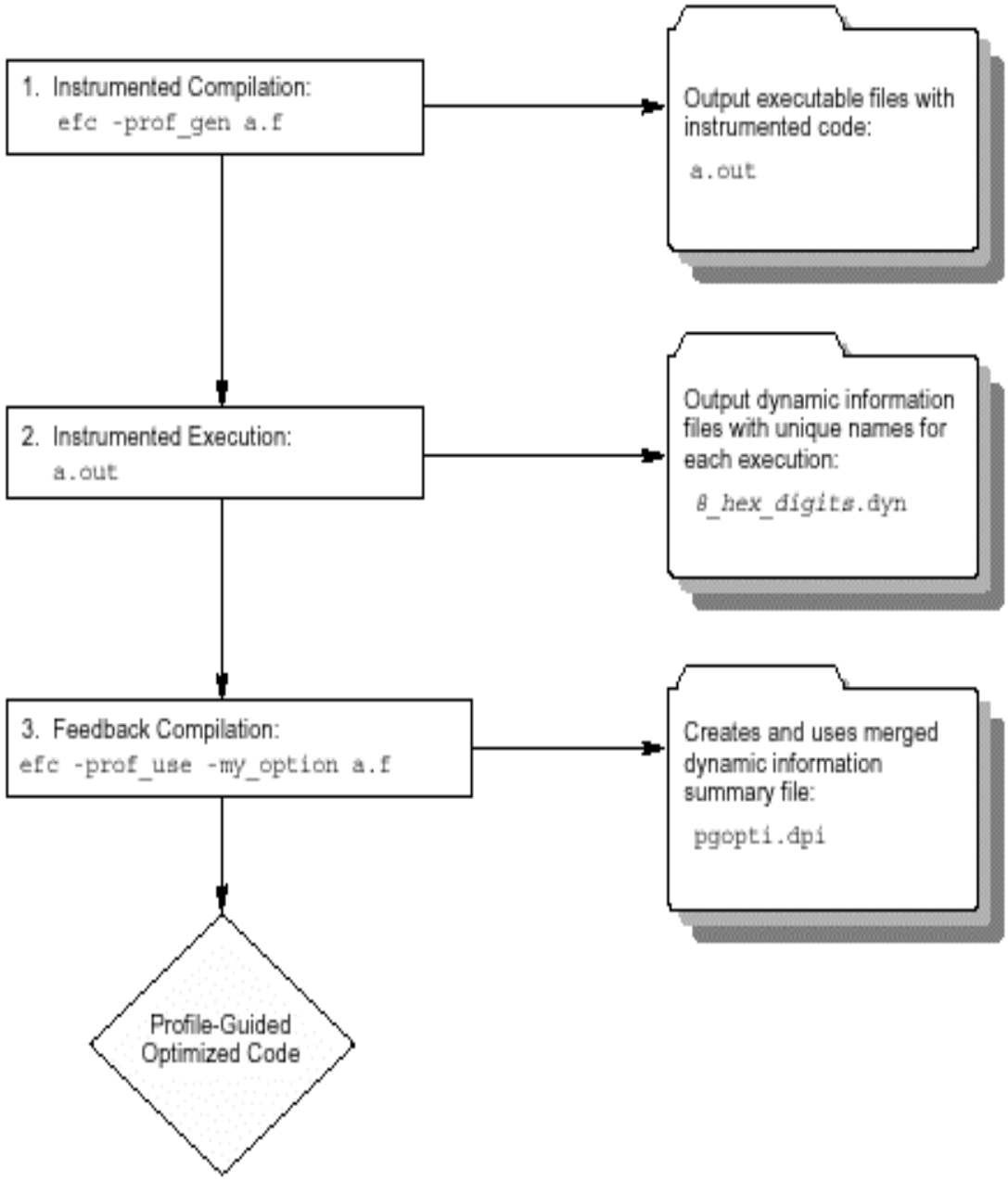
1. **Instrumentation compilation** and linking with `-prof_gen`
2. **Instrumented execution** by running the executable; as a result, the dynamic-information files (`.dyn`) are producedc
3. **Feedback compilation** with `-prof_use`

The flowcharts below illustrate this process for IA-32 compilation and Itanium(TM)-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(TM)-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. Used in phase 1 of the PGO to instruct the compiler to produce instrumented code in your object files in preparation for instrumented execution. With x qualifier, -prof_genx, gathers extra information for use with the Proforder tool.

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.

See an example of using PGO.

> **Note**
>
> For Itanium(TM)-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.

Advanced PGO Options

The options controlling advanced PGO optimizations are:

-prof_dir*dirname* and -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(TM)-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.

```
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 `-ip` in phase 3.

Note

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

The goal of function splitting is to improve the locality of executed instructions. Function splitting achieves this goal by splitting the non-executed code from the executed code. The executed code is emitted for each function, while the non-executed code is grouped together in a separate text section. See Basic PGO Options.

Function Order List

Function Order List Overview

A function order list is a text that specifies the order in which the linker should link the non-static functions of your program. This improves the performance of your program by reducing paging and improving code locality. Profile-guided optimizations support the generation of a function order list to be used by linker. The compiler determines the order using profile information.

Usage Guidelines

Use the following guidelines to create a function order list.

- The order list only affects the order of non-static functions.
- Do not use `-prof_genx` to compile two files from the same program simultaneously. This means that you cannot use the `-prof_genx` option with parallel makefile utilities.

Function Order List Utilities

To generate a function order list, the `profmerge` and `proforder` utilities are used.

The `profmerge` Utility

You will need to use the `profmerge` utility to merge the `.dyn` files.

This tool merges the dynamic profile information files (`.dyn`). The compiler executes this tool 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_dir*dirname*]**

Itanium(TM)-based applications:

prompt>**profmerge -em -p64 [-nologo] [-prof_dir*dirname*]**

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 compiler:

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

Itanium™ compiler:

prompt>**profmerge -em -p64 [-nologo] [-prof_file*filename*]**

where /prof_file*filename* is a profmerge utility option.

The proforder Utility

Use proforder to generate a function order list for use with the -ORDER linker option. The syntax for this tool is as follows:

prompt>**proforder [-prof_dir*dir_name*] [-o*order_file*]**

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

dir_name	is the directory containing the profile files (.dpi, .dyn, and .spi)
order_file	is the optional name of the function order list file. The default name is proford.txt

The proforder utility is used as part of the feedback compilation phase, to improve program performance.
To specify the filename of the .dpi file, use -prof_file*filename* option. The syntax for proforder with its -prof_file*filename* option is:

prompt>**proforder [-prof_file*filename*] [-o*orderfile*]**

Comparison of Function Order Lists and IPO Code Layout

The Intel® Fortran Compiler provides two methods of optimizing the layout of functions in the executable:

- use of a function order list
- use of -ipo

Each method has its advantages. A function order list, created with proforder, enables you to optimize the layout of non-static functions; that is, external and library functions whose names are exposed to the linker.
The compiler cannot affect the layout order for functions it does not compile, such as library functions. The function layout optimization is performed automatically when IPO is active.

Effects of the Function Order List		
Function Type	**Code Layout with -ipo**	**Function Ordering with proforder**
Extern	X	X

Library	No effect	X

Dump Profile Data Utility

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`.

Example of Function Order List Generation

Assume you have a Fortran program that consists of files `file1.f` and `file2.f` and that you have created a directory for the profile data files in `/usr/profdata`. Do the following to generate and use a function order list.

1. Compile your program by specifying `-prof_genx` and `-prof_dir`:

IA-32 compiler:

```
prompt>ifc -oMYPROG -prof_genx -prof_dir/usr/profdata
file1.f file2.f
```

Itanium(TM) compiler:

```
prompt>efc -oMYPROG -prof_genx -prof_dir/usr/profdata
file1.f file2.f
```

2. Run the instrumented program on one or more sets of input data.

```
prompt>MYPROG
```

The program produces a `.dyn` file each time it is executed.

3. Merge the data from one or more runs of the instrumented program using the `profmerge` tool to produce the `pgopti.dpi` file.

```
prompt>profmerge -prof_dir/usr/profdata
```

4. Generate the function order list using the proforder tool. By default, the function order list is produced in the file proford.txt.

```
prompt>proforder -prof_dir/usr/profdata -oMYPROG.txt
```

5. Compile your application with profile feedback by specifying the `-prof_use` and the `-ORDER` option to the linker. Again, use the `-prof_dir` option to specify the location of the profile files.

IA-32 compiler:

```
prompt>ifc -oMYPROG -prof_use -prof_dir/usr/profdata
file1.f file2.f -link -ORDER:@MYPROG.txt
```

Itanium compiler:

```
prompt>efc -oMYPROG -prof_use -prof_dir/usr/profdata
file1.f file2.f -link -ORDER:@MYPROG.txt
```

PGO API: Profile Information Generation Support

PGO API Support Overview

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)

HLO Overview

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(TM)-based applications. See Setting Optimization Levels.

IA-32 and Itanium(TM)-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(TM) 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` (`-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(TM) 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) Porcessor Optimization Reference Manual*.

Memory Dependency with IVDEP Directive

The `-ivdep_parallel` option discussed below is used for Itanium(TM)-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 inpoframtion on how to optimize with `-prefetch[-]`, refer to *Intel® Pentium® 4 and Intel® Xeon(TM) Porcessor Optimization Reference Manual*.

Parallelization

Parallelization Options Overview

For shared memory parallel programming, the Intel® Fortran Compiler supports both the OpenMP*, version 1.1 API, and an automatic parallelization capability. The following table lists the options that perform this support.

`-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. Default: `n`=75.
`-par_report{0|1|2|3}`	Controls the auto-parallelizer's diagnostic levels. Default: `-par_report1`.
`-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`

Note

If both `-openmp` and `-parallel` are specified on the command line, then:

- if and only if OpenMP directives are present within the subroutine, then the `-openmp` will be honored for this subroutine;
- else `-parallel` will be honored for this routine.

Auto-parallelization

The auto-parallelization feature of the Intel® Fortran Compiler implements a high-level symmetric multiprocessing (SMP) programming model that provides you with an easy way to exploit the parallelism of SMP systems.

Automatic parallelization relieves the user from having to deal with the low-level details of iteration modification, data partitioning, thread scheduling and synchronizations, while exploiting the performance potential available from multiprocessor systems.

Enabling Auto-parallelizer

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 multi-threaded code for these loops. An example of the command using auto-parallelization is as follows:

IA-32 compilations:

```
prompt>ifc -c -parallel -par_threshold0 myprog.f
```

Itanium-based compilations:

```
prompt>efc -c -parallel -par_threshold0 myprog.f
```

Auto-parallelization Environment Variables

Option	Description	Default
OMP_NUM_THREADS	Controls the number of threads used.	Number of processors currently installed in the system
OMP_SCHEDULE	Specifies the type of runtime scheduling.	static

Guidelines for Effective Auto-parallelization Usage

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 or global references.

Analyzing Compiler for Auto-parallelization

Currently, compiler is analyzed only on loop nests, but potentially on independent regions of code (task parallelism). A loop is parallelizable if:

- there is no loop-carried dependency or
- any loop-carried dependencies can be resolved by some code transformation, for example: privatization of scalars or runtime dependency testing.

  Privatization of scalars is an operation of re-assigning the storage of scalars from the static or parent stack area to the local stack of a thread to enable parallelization. This operation requires a WRITE permission and is usually performed to remove a data dependence between concurrently executing threads.

Preparing for Auto-parallelization

To prepare auto-parallelization, the compiler performs the following transformations:

- Partitions data accesses: shared, private, first-private, last-private, reduction
- Modifies loop parameters and references
- Generates new entry/exit per threaded task
- Generates both parallel and serial versions with conditional execution based on:
  - work/overhead threshold analysis
  - runtime dependency testing

Threshold for Auto-parallelization

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, n=0 to 100. 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.

Using Auto-parallelization

Autoparallelization enables you to automatically detect different code sequences which can be correctly and profitably executed concurrently, in separate threads. For example, the code in the first row of the following example can be computed in two threads:

```
do i=1,100
 a(i) = a(i) + b(i) * c(i)
enddo
```

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
```

Auto-parallelization Examples

Use auto-parallelization for the subroutine `daxpy`.

Subroutine daxpy

```
subroutine daxpy
 parameter (N=1000)
 double precision a(N), b(N), s
   do i=1,N
      a(i) = a(i) * s + b(i)
   enddo
 end
```

Parent Thread Code

```
subroutine daxpy
 parameter (N=1000)
 double precision a(N), b(N), s
 external daxpy_par_loop0, kmpc_ok_to_fork
```

```
 logical kmpc_ok_to_fork
   if ( kmpc_ok_to_fork() .eqv. .true. ) then
         call kmpc_fork_call( daxpy_par_loop0, a, b, s )
   else
         call kmpc_serialized_parallel( kmpc_tid )
             call daxpy_par_loop0( kmpc_tid, a, b, s )
         call kmpc_end_serialized_parallel( kmpc_tid )
   endif
 end
```

In the above example, work threshold and runtime dependency tests are combined in this `if` statement:

```
if ( kmpc_ok_to_fork() .eqv. .true. )
```

Child Thread Code

```
subroutine daxpy(kmpc_tid, a, b, s)
 parameter (N=1000)
 double precision a(N), b(N), s
 integer lb, ub, inc
   call kmpc_for_static_init( kmpc_tid, lb, ub, inc )
     do i=lb,ub,inc
        a(i) = a(i) * s + b(i)
     enddo
   call kmpc_for_static_fini( kmpc_tid )
 end
```

Using /Qpar_report

```
subroutine daxpy()
 parameter (N=1000)
 external foo
 integer, pointer :: q(:)
 C Assumed side effects
   do i=1,N
      q(i) = q(i) + foo()
   enddo
 C Real dependency
   do i=1,N
      q(i) = q(i-1) + i
   enddo
 end
```

Using Auto-parallelization Directive

```
subroutine daxpy
 parameter (N=1000)
 double precision a(N)
 integer x(N)
 !DIR$ PARALLEL
  do i=1,N
     a(i) = a(x(i)) * 2
  enddo
 end
```

Auto-parallelizer's Diagnostic

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).

`-par_report2` = indicates loops successfully and unsuccessfully auto-parallelized.

`-par_report3` = same as 2 plus additional information about any proven or assumed dependences inhibiting auto-parallelization.

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 (see more on this directive later in this topic).
- Use `-ipo` to eliminate assumed side-effects

Example below shows an aoutput generated by `-par_report3` as a result from the command:

```
prompt>ifc -c -parallel -par_threshold0 -par_report3
myprog.f
```

-par_report Output Example

```
external subroutine XYZ
 PROCEDURE: XYZ
 SERIAL LOOP: line 9: non-parallel candidate loop
due to statement at line 10
 SERIAL LOOP: Line 14
 Reason:  FLOW dependence for "q" from line 15 to
line 15.

 18 Lines Compiled
```

!DIR$PARALLEL Directive

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.

Parallelization with OpenMP*

For shared memory parallel programming, the Intel® Fortran Compiler supports the OpenMP*, version 1.1 API. The OpenMP Fortran API has recently emerged as a standard for shared memory parallel programming. This feature relieves the user from having to deal with the low-level details of iteration partitioning, data sharing, and thread scheduling and synchronization. It also provides the benefit of the performance available from multiprocessor systems.

The Intel® Fortran Compiler supports OpenMP API version 1.1 and performs code transformation to automatically generate multithreaded codes based on the user's OpenMP directive annotations

in the program. For more information on the OpenMP standard, visit www.openmp.org web site. The Intel Extensions to OpenMP topic describes the extensions to the version 1.1 standard that have been added by Intel in the Intel Fortran Compiler.

Note

As with many advanced features of compilers, you must be sure to properly understand the functionality of the auto-parallelization options in order to use them effectively and avoid unwanted program behavior.

Command Line Options

The Parallelization capability of the Intel Fortran Compiler uses the following options:

Option	Description	Default
`-openmp`	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.	OFF
`-openmp _report{0|1|2}`	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 (default). `-openmp_report2` = same as `-openmp_report1` plus diagnostics indicating master construct, single construct, critical sections, order construct, atomic directive, etc. successfully handled.	`-openmp_ report1`

OpenMP* Standard Option

For complete information on the OpenMP* standard, visit the www.openmp.org web site. The Intel Extensions to OpenMP topic describes the extensions to the standard that have been added by Intel in the Intel® Fortran Compiler.

OpenMP Fortran Directives and Clauses

An OpenMP directive has the form:

```
omp-sentinel directive [directive clause [ directive clause
. . .]
```

An `omp-sentinel` is either

```
!$OMP
C$OMP
```

with no intervening spaces for fixed form source input, or

```
!$OMP
```

for free form source input.

OpenMP Environment Variables

Variable	Description	Default
OMP_SCHEDULE	Sets the run-time schedule type and chunk size.	STATIC
OMP_NUM_THREADS	Sets the number of threads to use during execution.	Number of processors
OMP_DYNAMIC	Enables or disables the dynamic adjustment of the number of threads.	.FALSE.
OMP_NESTED	Enables or disables nested parallelism.	.FALSE.

See the lists of OpenMP* Standard Directives and Clauses in the Reference section.

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.

See the List of OpenMP* Runtime Library Routines in the Reference section.

Intel Extensions to OpenMP*

This topic describes the extensions to the standard that have been added by Intel in the Intel® Fortran Compiler. For complete information on the OpenMP* standard, visit the www.openmp.org website.

Environment Variables

Environment Variable	Description
KMP_STACKSIZE	Gets and sets the wait time in milliseconds that the libraries wait after completing the execution of a parallel region before putting threads to sleep.
KMP_BLOCKTIME	Gets and sets the number of bytes to allocate for each parallel thread to use asits private stack.
KMP_SPIN_COUNT	Helps to fine-tune the critical section.

Thread-level MALLOC( )

The Intel Fortran Compiler implements an extension to the OpenMP runtime library to enable threads to allocate memory from a heap local to each thread.

The memory allocated by these routines must also be freed by the FREE routine. While it is legal for the memory to be allocated by one thread and FREE'd by a different thread, this mode of operation has a slight performance penalty.

The interface is identical to the MALLOC( ) interface except the entry points are prefixed with KMP_, as shown below.

Prototype

```
INTERFACE
 INTEGER FUNCTION KMP_MALLOC
(KMP_SIZE_t)
```

```
 INTEGER KMP_SIZE_t
 END FUNCTION KMP_MALLOC
 END INTERFACE
```

KMP_SIZE_t is the number of bytes of memory to be allocated

```
INTERFACE
 SUBROUTINE
KMP_FREE(KMP_ADDRESS)
 INTEGER KMP_ADDRESS
 END SUBROUTINE KMP_FREE
 END INTERFACE
```

KMP_ADDRESS is the starting address of the memory block to be freed.

Examples of OpenMP* Usage

The following examples show how to use the OpenMP* feature.

A Simple Difference Operator

This example shows a simple parallel loop where each iteration contains 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
```

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
```

Vectorization (IA-32 Only)

Vectorization Overview

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 program to process 2, 4, 8 or up to 16 elements in one operation, depending on the data type.

This section provides options description, guidelines, and examples for Intel® Fortran Compiler vectorization implemented by IA-32 compiler only. The following list summarizes this section contents.

- A quick reference of vectorization functionality and features
- Descriptions of compiler options to control vectorization
- Vectorization Key Programming Guidelines
- Descriptions of the Fortran language features to control vectorization
- Discussion and general guidelines on vectorization levels:
  - automatic vectorization
  - vectorization with user intervention
- Examples demonstrating typical vectorization issues and resolutions

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.

`-x{M|K|W}`	Generate specialized code to run exclusively on the processors supporting 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, on a single binary, code specialized to the extensions specified by `{M|K|W}` but also generates 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 as follows:
	`n` = 0: no information
	`n` = 1: indicates vectorized /non-vectorizerd integer loops
	`n` = 2: indicates vectorized /non-vectorized integer loops
	`n` = 3: indicates vectorized /non-vectorized integer loops and prohibit data dependence information
	`n` = 4: indicates non-vectorized loops
	`n` = 5: indicates non-vectorized loops and prohibit data dependence information

`-vec-`	Turns off vectorizer.

Vectorization Key Programming Guidelines

The goal of vectorizing compilers is to exploit single-instruction multiple data (SIMD) processing automatically. 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. However, you should make only the changes needed to enable vectorization and no others.

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 assignments. Array references can appear on the left hand side of assignments.
- Only assignment statements

Avoid:

- Function calls
- Unvectorizable operations
- 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 two memory references at 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 vectorizations. 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.

An example where data dependencies prohibit vectorization is shown below. In this example, each element of an array is changed to be function of itself and its two neighbors.

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, the iterations must be done 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)
  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 if the final stored value is a 16-bit integer. Because the MMX(TM) and SSE instruction sets are not fully orthogonal (byte shifts, 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.

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.
   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 obove.

Language Support and Directives

This section addresses language features that enhance vectorization of application code.

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. 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, reducing the recurrence II.

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(TM)-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$VECTOR ALWAYS
 !DIR$VECTOR ALIGNED
 !DIR$VECTOR UNALIGNED
```

The VECTOR ALWAYS Directive

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
```

The VECTOR ALIGNED/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.

The NOVECTOR Directive

If, on the other hand, avoiding vectorization of a loop is more reasonable than using it (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 desired, the NOVECTOR directive can be used, as shown in the example that follows.

NOVECTOR

```
!DIR$ NOVECTOR
      do i = 1, 100
       a(i) = b(i) + c(i)
      enddo
```

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

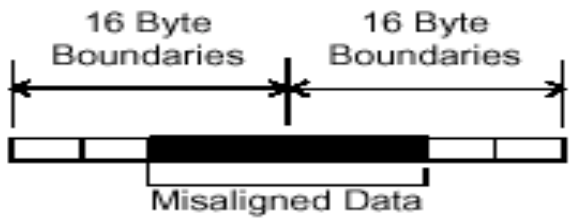
```
SUBROUTINE
VEC_COPY(DEST,A,B,LEN)
 DIMENSION DEST(*)
 INTEGER A(*), B(*)
 INTEGER LEN, I
 DO I=1,LEN
   DEST(A(I)) = DEST(B(I))
 END DO
RETURN
END
```

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.

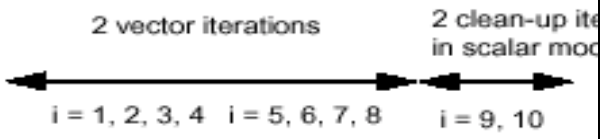
The Misaligned Data Crossing 16-Byte Boundary figure shows the effect of a data cache unit (DCU) split due to misaligned data. The code loads the misaligned data across a 16-byte boundary, which results in an additional memory access causing a six- to twelve-cycle stall. You can avoid the stalls if you 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 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.

Optimizer Report Generation (Itanium(TM) Compiler)

The Intel® Fortran Itanium(TM) Compiler for Itanium(TM)-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`	Electron Code Generator
`omp`	Open MP
`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`	Electron Code Generator, software pipelining

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 availble for report generation.

Libraries

Managing 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(TM) 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(TM)-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` libary 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(TM)-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(TM) 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++ 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 Dynamic Shared Object (DSO) instead of an executable. For more details, refer to the `ld` man page documentation.

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(TM)-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.

Math Libraries

Math Libraries Overview

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(TM) 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(TM) Compiler, libm.a is optimized for for the use with Itanium architecture.

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 Forrtran 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*.

Diagnostics and Messages

Diagnostics Overview

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 (IA-32 Compiler Only)

Runtime Diagnostics Overview

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.

Generating Diagnostic Reports

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.
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
```

Messages and Obtaining Information

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(TM) compiler:
	`prompt>efc -help` or `prompt>efc -?`

Showing compiler version and driver tool commands	
`-V`*`string`*	Displays compiler version information. Puts *`string`* in the .comment section of the object file.
`-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.
-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(TM) 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 module 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.

`-s`	Enables/disables 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

Mixing C and Fortran Overview

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 Microsoft* attributes 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

OpenMP* Reference Information

List of OpenMP* Standard Directives and Clauses

OpenMP* Directives

Directive	Description
`parallel`	Defines a parallel region.
`do, enddo[nowait]`	Identifies an iterative work-sharing construct that specifies a region in which the iterations of the associated loop should be executed in parallel. The argument `nowait` indicates that the loop that reached the end can proceed with further execution on its thread. If `nowait` is absent, all loops have to reach the end, and only then the execution continues on all threads.
`sections`	Identifies a non-iterative work-sharing constuct that specifies a set of constucts that are to be divided among threads in a team.
`section`	Indicates that the associated code block should be executed in parallel.
`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` 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 other statement or an OpenMP directive between the `parallel` or `do` directive and the `do` statement, the Intel® Fortran compiler issues a syntax error.
`parallel sections`	Provides a shortcut form for specifying a parallel region containing a single sections directive.
`master, end master`	Identifies a constuct that specifies a structured block that is executed by 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.
`barrier`	Synchronizes all the threads in a team.

`atomic`	Ensures that a specific memory location is updated atomically.
`flush`	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.
`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`	Makes the named file-scope or namespace-scope variables specified private to a thread but file-scope visible within the thread.

OpenMP Clauses

Clause	Description
`private`	Declares variables to be `private` to each thread in a team.
`firstprivate`	Provides a superset of the functionality provided by the `private` clause.
`lastprivate`	Provides a superset of the functionality provided by the `private` clause.
`shared`	Shares variables among all the threads in a team.
`default`	Enables you to affect the data-scope attributes of variables.
`reduction`	Performs a reduction on scalar variables.
`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.
`if`	If `IF(scalar_logical_expression)` clause is present, the enclosed code block is executed in parallel only if the `scalar_logical_expression` evaluates to `.TRUE.`. Otherwise the code block is serialized.
`schedule`	Specifies how iterations of the `do` loop are divided among the threads of the team.
`copyin`	Provides a mechanism to assign the same name to `threadprivate` variables for each thread in the team executing the parallel region.

List of OpenMP* Runtime Library Routines

The following table specifies the interface to OpenMP* runtime library routines. The names for the routines are in user name space. The omp.h header file is provided in the include directory of your compiler installation. There are definitions for two different locks, omp_lock_t and omp_nest_lock_t, which are used by the functions in the table.

Function	Description
`SUBROUTINE omp_set_num_threads(num_thr`	Sets the number of threads to use for subsequent parallel regions.

`omp_set_num_threads(num_threads)`	
`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 on the current machine.
`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) INTEGER dynamic_threads`	Enables or disables dynamic adjustment of the number of threads used to execute a parallel region. If `dynamic_threads` is non-zero, dynamic threads are enabled. If `dynamic_threads` is zero, dynamic threads are disabled.
`LOGICAL 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 parameter is non-zero, enable. Default is disabled.
`LOGICAL FUNCTION omp_get_nested()`	Determines whether nested parallelism is currently enabled or disabled. Function returns non-zero if nested parallelism is supported, zero otherwise. Always returns `.FALSE.` in the current version of compiler.
`SUBROUTINE omp_init_lock(lock) INTEGER lock`	Initializes a unique `lock` and sets `lock` to its value.
`SUBROUTINE omp_destroy_lock(lock)` `INTEGER lock`	Disassociates `lock` from any `lock`s.
`SUBROUTINE omp_set_lock(lock)` `INTEGER 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)`	Releases executing thread from ownership of `lock` associated with `lock`. The `lock` argument must be initialized via

`INTEGER lock`	`omp_init_lock()`, and behavior undefined if executing thread does not own the `lock` associated with `lock`.
`INTEGER omp_test_lock(lock)`	Attempts to set `lock` associated with `lock`. If successful, returns non-zero. `lock` must be initialized via `omp_init_lock(lock)`.
`SUBROUTINE omp_init_nest_lock(lock)` `INTEGER lock`	Initializes a unique nested `lock` and sets `lock` to its value.
`SUBROUTINE omp_destroy_nest_lock(lock)` `INTEGER lock`	Disassociates the nested lock "`lock`" from any `lock`s.
`SUBROUTINE omp_set_nest_lock(lock)` `INTEGER 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_nest_lock(lock)` `INTEGER lock`	Releases executing thread from ownership of `lock` associated with `lock` if count is zero. `lock` must be initialized via `omp_init_nest_lock()`. Behavior is undefined if executing thread does not own the `lock` associated with `lock`.
`INTEGER omp_test_nest_lock(lock)`	Attempts to set `lock` associated with `lock`. If successful, returns nesting count, otherwise returns zero. `lock` must be initialized via `omp_init_lock()`.

Compiler Limits

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

Additional Intrinsic Functions Overview

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.

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.

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.

%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.
- If a function name is used as an actual argument, then it must be a specific name, not a generic name.
- If a function name is used as a dummy argument, then it does not identify an intrinsic function in the subprogram, but has a data type according to the normal rules for variables and arrays.

Additional Intrinsic Functions

Intrinsic Function	Definition	Generic Name	Specific Name	No. of Args	Type of Args	Type of Function
Type conversion	Conversion to double precision See Note 1	DREAL		1	real real*16 doubl complex*32	real real*16 double complex*32
		DFLOAT		1	integer*2 integer*4 integer*8	real*8 real*8 real*8
	Conversion to dou complexSee Note	DCMPLX		1 or 2	integer*2 integer*4 integer*8 real*4 real*8 real*16 real*16 complex*8 complex*16 complex*32 complex*32	complex*16 complex*16 complex*16 complex*16 complex*16 complex*16 complex*16 complex*16 complex*16 complex*16 complex*32
Absolute value	|x|	ABS	ZABS CDABS TABS DABS QABS	1	dcomplex dcomplex real double real*16 complex*32	double double real double real*16 complex*32
Imaginary part of a complex argument	xi	IMAG	DIMAG CDIMAG TIMAG QIMAG	1	dcomplex dcomplex real real*16 complex*32	double double real real*16 complex*32
Conjugate of a complex argument of a complex argument	(xr, -xi)	CONJG	DCONJ GTCONJ DCONJ QCONJ	1	dcomplex real double complex*32	double real double complex*32
Square root	Ðx	SQRT	ZSQRT SQRT TSQRT DSQRT	1	dcomplex dcomplex real real*16	dcomplex dcomplex real real*16
Exponential	ex	EXP	ZEXP CDEX TEXP QEXP DEXP	1	dcomplex dcomplex real double real*16 double	dcomplex dcomplex real double complex*32 double
Natural Logarithm	loge(x)	LOG	ZLOG CDLOG DLOG QLOG	1	dcomplex dcomplex real*16 real*16 complex*32	dcomplex dcomplex double real*16 complex*32

Bitwise Operation	AND		AND	2	integer	integer
See Note 1	OR		OR	2	integer	integer
	Exclusive OR		XOR	2	integer	integer
	Shift left: x1 logica shifted left x2 bits. must be > 0		LSHIFT	2	integer	integer
	Shift right: x1 logic shifted right x2 bits must be > 0		RSHIFT	2	integer	integer
Environ-mental Inquiries. See Note 1	Base of number systems		EPBASE	1	real double real*16 real*16 complex*32	integer integer integer integer complex*32
	Number of Signific Bits		EPPREC	1	real double real*16 real*16 complex*32	integer integer integer integer integer
	Minimum Exponen		EPEMIN	1	real double real*16 real*16 complex*32	integer integer integer integer integer
	Maximum Exponent		EPEMAX	1	real double real*16 real*16 complex*32	integer integer integer integer integer
	Smallest non-zero number		EPTINY	1	real double real*16 double complex*32	real double real*16 double double
	Largest Number Representable		EPHUGE	1	integer real double real*16 double complex*32	integer real double real*16 double double
	Epsilon		EPMRSP	1	real double real*16 double complex*32	real double real*16 double complex*32
Location See Note 3	Address of	LOC		1	any	integer
Sine	sin(x)	SIN SIND	ZSIN SIND DSIND QSIND	1	dcomplex real*16 double real*16	dcomplex real*16 double real*16

					complex*32	complex*32
Cosine	cos(x)	COS COSD	ZCOS CDCOS COSD DCOSD QCOSD	1	dcomplex dcomplex real double real*16 complex*32	dcomplex dcomplex real double real*16 complex*32
Tangent	tan(x)	TAND	TAND DTAND QTAND	1	real double real*16 complex*32	real double real*16 complex*32
Arcsine	arcsin(x)	ASIND	ASIND DASIND QASIND	1	real double real*16 complex*32	real double real*16 complex*32
Arc-cosine		ACOSD	ACOSD QCOSD DACOSD QACOSD	1	real complex*32 double real*16 complex*32	real complex*32 double real*16 complex*32
Arctangent	arctan(x)	ATAND	ATAND DATAND QATAND	1	real double real*16 complex*32	real double real*16 complex*32
	arctan(x1-x2)	ATAN2D	ATAN2D DATAN2 XATAN2 QATAN2	2222	real double real*16 real*16 complex*32	real double real*16 real*16 complex*32

Intel Fortran Compiler Key Files

Key Files Summary for IA-32 Compiler

The following tables list and briefly describe files that are installed for use by the IA-32 version of the compiler.

/bin Files

File	Description
`f90com`	Executable used by the compiler
`fpp`	Fortran preprocessor
`ifc`	Intel® Fortran Compiler

ifc.cfg	Configuration file for use from command line
ifc.pcl	Default program catalog list of work.pc files to search for module references
ifccem	FCE Manager Utility
ifcvars.csh	Environment variables header file
ifcvars.sh	Batch file to set environment variables
profmerge	Utility used for Profile Guided Optimizations
proforder	Utility used for Profile Guided Optimizations
xiar	Tool used for final interprocedural compilation prior to archiving.
xild	Tool used for Interprocedural Optimizations

/lib Files

File	Description
libbindf90.a	Library of Binder utilities
libcepcf90.a	Fortran I/O library to coexist with C
libcepcf90.so	Shared Fortran I/O library to coexist with C
lincprts.a	C++ standard language library
lincprts.so	Shared C++ standard language library
libcxa.a	C++ language library indicating I/O data location
libcxa.so	Shared C++ language library indicating I/O data location
libf90.a	Intel-specific Fortran runtime library
libf90.a	Shared Intel-specific Fortran runtime library
libguide.a	OpenMP library
libguide.so	Shared OpenMP library
libiepcf90.a	Intel-specific Fortran runtime I/O library
libiepcf90.so	Shared Intel-specific Fortran runtime I/O library

libimf.a	Special purpose math library functions, including some transcendentals, built only for Linux
libimf.so	Shared special purpose math library functions, including some transcendentals, built only for Linux
libintrins.a	Intrinsic functions library
libintrins.so	Shared intrinsic functions library
libirc.a	Intel-specific library (optimizations)
libompstub.a	Library to resolve references to OpenMP subroutines when OpenMP is not used
libpepcf90.a	Portability library
libpepcf90.so	Shared portability library
libposf90.a	Posix library
libposf90.a	Shared posix library
libsvml.a	Short-vector math library (used by vectorizer)
libunwind.a	Exception handling library to perform stack unwinds
libunwind.so	Shared version of exception handling library

Key Files Summary for Itanium(TM) Compiler

The following tables list and briefly describe files that are installed for use by the Itanium(TM) compiler version of the compiler.

/bin Files

File	Description
f90com	Executable used by the compiler
fpp	Fortran preprocessor
efc	Intel® Fortran Compiler
efc.cfg	Configuration file for use from command line
efc.pcl	Default list of work.pc files to search for Fortran module references
efccem	FCE Manager Utility

efcvars.csh	Environment variables header file
efcvars.sh	Batch file to set environment variables
profmerge	Utility used for Profile Guided Optimizations
proforder	Utility used for Profile Guided Optimizations
xiar	Tool used for final interprocedural compilation prior to archiving.
xild	Tool used for Interprocedural Optimizations

/lib Files

File	Description
libasmutils.so	Library of Intel Itanium Assembler utilities
libcepcf90.a	Fortran I/O library to coexist with C
libcepcf90.so	Shared Fortran I/O library to coexist with C
libcprts.a	C++ standard language library
libcprts.so	Shared C++ standard language library
libcxa.a	C++ language library indicating I/O data location
libcxa.so	Shared C++ language library indicating I/O data location
libdeceia.a	Assembler decoder library for IA-32 instructions on Itanium processor.
libdeceia.so	Shared assembler decoder library for IA-32 instructions on Itanium processor.
libdecem.a	Assembler decoder library for Itanium processor.
libdecem.so	Shared assembler decoder library for Itanium processor.
libdecem68.a	Assembler decoder library for Pentium® 4 processor.
libdecem68.so	Shared assembler decoder library for Pentium® 4 processor.
libdiseia.a	Disassembly library for IA-32 instructions on Itanium processor.
libdiseia.so	Shared disassembly library for IA-32 instructions on Itanium processor..
libdisem.a	Disassembly library for Itanium processor.

libdisem.so	Shared disassembly library for Itanium processor..
libdisp68.a	Disassembly library for Pentium 4 processor.
libdisp68.so	Shared disassembly library for Pentium 4 processor.
libenceia.a	Assembler encoder library for IA-32 instructions on Itanium processor.
libenceia.so	Shared assembler encoder library for IA-32 instructions on Itanium processor.
libencem.a	Assembler encoder library for Itanium processor.
libencem.so	Shared assembler encoder library for Itanium processor.
ibencp68.a	Assembler encoder library for Pentium 4 processor.
libencp68.so	Shared assembler encoder library for Pentium 4 processor
libf90.a	Intel-specific Fortran run-time library
libf90.so	Shared Intel-specific Fortran run-time library
libfpel.a	Floating point emulation assembly library.
libguide.a	OpenMP library
libguide.so	Shared OpenMP library
libiel.a	Integer emulation assembly library.
libiepcf90.a	Intel-specific Fortran I/O library
libiepcf90.so	Shared Intel-specific Fortran I/O library
libiline.so	Assembly library.
libimf.a	Intel special purpose math library functions, including some transcendentals.
libintrins.a	Intrinsic functions library
libintrins.so	Shared intrinsic functions library
libirc.a	Intel-specific library (optimizations)
libm.a	Math library compatible with GNU.
libmofl.a	Multiple Object Format Library, used by the Intel assembler

libmofl.so	Shared Multiple Object Format Library, used by the Intel assembler
libpepcf90.a	Portability library
libpepcf90.so	Shared portability library
libposf90.a	Posix library
libposf90.so	Shared posix library
libsched.so	Shared assembly scheduling library
libsymdbg.so	Shared assembly symbolic debugger library
libunwdecem.a	Assembly decoder exception handling library to perform stack unwinds
libunwdecem.so	Shared assembly decoder exception handling library to perform stack unwinds
libunwind.a	Exception handling library to perform stack unwinds
libunwind.so	Shared exception handling library to perform stack unwinds
libvral.so	Assembly virtual register allocation library

Lists of Error Messages

Error Message Lists Overview

This section provides lists of error messages generated during compilation phases or reporting program error conditions. It includes the error messages for the following areas:

- runtime
- allocation
- input-output
- intrinsic procedures
- mathematical
- exceptions

Runtime Errors (IA-32 Only)

These errors are caused by an invalid run-time operation. Following the message, a post-mortem report is printed if any of the compile-time options `-C, -CA, -CB, -CS, -CU, -CV` or `-d{n}` was selected.

Error	Option(s) Required	Message
401	`-CU`	Unassigned variable

404	none	Assigned label is not in specified list
405	none	Integer is not assigned with a format label
406	`-CB`	Array bounds exceeded
439	none	nth argument is not present
440	none	Inconsistent lengths in a pointer assignment
442	none	Inconsistent length for `CHARACTER` pointer function
*447	`-CS`	Invalid `DIM` argument to `LBOUND`
*448	`-CS`	Invalid `DIM` argument to `UBOUND`
*449	`-CS`	Invalid `DIM` argument to `SIZE`
451	none	Procedure is a `BLOCKDATA`
454	`-CS`	Array shape mismatch
455	`-CB`	Array section bounds inconsistent with parent array
456	`-CB`	Invalid `character` substring ending position
457	`-CB`	Invalid `character` substring ending position
458	none	Object not allocated
459	`-CA`	Array not allocated
460	`-CA`	Pointer not allocated
461	`-CA, -CU`	Pointer is undefined
462	`-CA`	Assumed-shape array is not allocated
463	`-CA`	Assumed-shape array is undefined
464	none	Inconsistent lengths in a `character` array constructor 441 `-CV` 443 `-CV` 444 `-CV` 480 `-CV` 481 `-CV`
441	`-CV`	Inconsistent length for `CHARACTER` pointer argument argument-name
443	`-CV`	Inconsistent length for `CHARACTER` argument
444	`-CV`	Inconsistent length for `CHARACTER` function
480	`-CV`	Too many arguments specified

481	-CV	Not enough arguments specified
*482	-CV	Incorrect interface block
*483	-CV	Interface block required for subprogram-name
*484	-CV	name is not a type-kind function-subroutine
*485	-CV	Argument type mismatch
*486	-CV	Array rank mismatch

*These errors are followed by additional information, as appropriate:

- nth dummy argument is not an actual-argument-type
- type1 actual argument passed to type2 dummy argument n
- type actual argument passed to cray-pointer dummy argument n
- Cray-pointer actual argument passed to type dummy argument n
- nth dummy argument is [not] a cray-pointer
- nth actual argument is not compatible with type RECORD
- name is [not] a pointer-valued function
- nth dummy argument is [not] a pointer
- name is [not] a dynamic CHARACTER function
- nth dummy argument is [not] optional
- nth dummy argument is [not] an assumed-shape array
- name is [not] an array-valued function
- nth dummy argument is an array but the actual argument is a scalar
- nth dummy argument is a scalar but the actual argument is an array
- The actual rank (x) of name does not match the declared rank (y)
- The data type of name does not match its declared type
- nth dummy argument and the actual argument are different data types
- nth actual argument passed to Fortran subprogram using %VAL
- nth actual argument passed to Fortran subprogram using %REF

Allocation Errors

The following errors can arise during allocation or deallocation of data space.

If the relevant ALLOCATE or DEALLOCATE includes a STAT = specifier, then an occurrence of any of the errors below will cause the STAT variable to become defined with the corresponding error number, instead of the error message being produced.

In the error messages, vartype is

array	a pointer to an array, an allocatable array, or a temporary array
character scalar	a pointer to a character scalar, an automatic character scalar, or a temporary character scalar
pointer	a pointer to a non-character scalar

Error	Message
491	vartype is already allocated.
492	vartype is not allocated.
493	vartype was not created by ALLOCATE.
494	Allocation of nnn bytes failed or Allocation of array with extent nnn failed or Allocation of array with element size nnn failed or Allocation of character scalar with element size nnn failed or Allocation of pointer with element size nnn failed.
495	Heap initialization failed.

Input/Output Errors

The number and text of each input-output error message is given below, with the context in which it could occur and an explanation of the fault which has occurred. If the input-output statement includes an IOSTAT=STAT specifier, then an occurrence of any of the errors that follow will cause the STAT variable to become defined with the corresponding error number.

Error	Message	Where Occurring	Description
117	Unit not connected	OPEN	An attempt was made to read or write to a closed unit.
118	File already connected	OPEN	An attempt was made to OPEN a file on one unit while it was still connected to another.
119	ACCESS conflict	OPEN, Positional, READ, WRITE	When a file is to be connected to a unit to which it is already connected, then only the BLANK, DELIM, ERR, IOSTAT and PAD specifiers may be redefined. An attempt has been made to redefine the ACCESS specifier. This message is also used if an attempt is made to use a direct-access I/O statement on a unit which is connected for sequential I/O or a sequential I/O statement on a unit connected for direct access I/O.
120	RECL conflict	OPEN	When a file is to be connected to a unit to which it is already connected, then only the BLANK, DELIM, ERR, IOSTAT and PAD specifiers may be redefined. An attempt has been made to redefine the RECL specifier.
121	FORM conflict	OPEN	When a file is to be connected to a unit to which it is already connected, then only the BLANK, DELIM, ERR, IOSTAT and PAD specifiers may be redefined. An attempt has been made to redefine the FORM specifier.
122	STATUS conflict	OPEN	When a file is to be connected to a unit to which it is already connected, then only the BLANK, DELIM, ERR, IOSTAT and PAD specifier may be redefined. An attempt has been made to redefine the STATUS specifier.

123	Invalid STATUS	CLOSE	STATUS=DELETE has been specified in a CLOSE statement for a unit which has no write permissions; for example, the unit has been opened with the READONLY specifier.
125	Specifier not recognized	OPEN	A specifier value defined by the user has not been recognized.
126	Specifiers inconsistent	OPEN	Within an OPEN statement one of the following invalid combinations of specifiers was defined by the user: ACCESS=DIRECT was specified when STATUS=APPEND BLANK=FORMATTED was specified when FORM= UNFORMATTED
127	Invalid RECL value	OPEN, DEFINE FILE	The value of the RECL specifier was not a positive integer.
128	Invalid filename	INQUIRE	The name of the file in an Inquire by file statement is not a valid filename.
129	No filename specified	OPEN	In an OPEN statement, the STATUS specifier was not SCRATCH or UNKNOWN and no filename was defined.
130	Record length not specified	OPEN	The RECL specifier was not defined although ACCESS=DIRECT was specified.
131	An equals expected	Namelist READ	A variable name, array element or character substring reference in the input was not followed by an '='.
132	Value separator missing	List-Directed READ, Namelist READ	A complex or literal constant in the input stream was not terminated by a delimiter (that is, by a space, a comma or a record boundary).
133	Value separator expected	Namelist READ	A subscript value in a character substring or array element reference in the input was not followed by a comma or close bracket.
134	Invalid scaling	WRITE with FORMAT	If d represents the decimal field of a format descriptor and k represents the current scale factor, then the ANSI Standard requires that the relationship - d<k<d+2 is true when an E or D format code is used with a WRITE statement. This requirement has been violated.
135	Invalid logical value	Formatted READ	A logical value in the input stream was syntactically incorrect.
136	Invalid character value	Namelist READ	A character constant does not begin with a quote character.
137	Value not recognized	List-Directed READ, Namelist READ	An item in the input stream was not recognized.
138	Invalid repetition value	List-Directed READ, Namelist READ	The value of a repetition factor found in the input stream is not a positive integer constant.
139	Illegal repetition factor	List-Directed READ, Namelist READ	A repetition factor in the input stream was immediately followed by another repetition factor.
140	Invalid integer	Formatted READ	The current input field contained a real number when

			an integer was expected.
141	Invalid real	Formatted READ	The current input field contained a real number which was syntactically incorrect.
143	Invalid complex constant	List-Directed READ, Namelist READ	The current input field contained a complex number which was syntactically incorrect.
144	Invalid subscript	Namelist READ	A subscript value in an array element reference in the input was not a valid integer.
145	Invalid substring	Namelist READ	A subscript value in a character substring reference was not a valid integer or was not positive.
146	Variable not in Namelist	Namelist READ	The data contained an assignment to a variable which is not in the NAMELIST list.
147	Variable not an array	Namelist READ	A variable name in the data was followed by an open bracket but the name is not an array or character variable.
148	Invalid character	Formatted READ	A character has been found in the current input stream which cannot syntactically be part of the entity being assembled.
149	Invalid Namelist input	Namelist READ	The first character of a record read by a Namelist READstatement was not a space.
150	Literal not terminated	List-Directed READ, Namelist READ	A literal constant in the input file was not terminated by a closing quote before the end of the file.
151	A variable name expected	Namelist READ	A list of array or array element values in the data contained too many values for the associated variable.
152	File does not exist	OPEN	An attempt has been made to open a file which does not exist with STATUS=OLD.
153	Input file ended	READ	All the data in the associated internal or external file has been read.
154	Wrong length record	READ, WRITE	The record length as defined by a FORMAT statement, or implied by an unformatted READ or WRITE, exceeds the defined maximum for the current input or output file.
155	Incompatible format descriptor	READ/WRITE with FORMAT	A format description was found to be incompatible with the corresponding item in the I-O list.
156	READ after WRITE	READ	An attempt has been made to read a record from a sequential file after a WRITE statement.
158	Record number out of range	Direct Access READ/WRITE, FIND	The record number in a direct-access I-O statement is not a positive value, or, when reading, is beyond the end of the file.
159	No format descriptor for data item	READ/WRITE with FORMAT	No corresponding format code exists in a FORMAT statement for an item in the I-O list of a READ or WRITE statement.
160	READ after Endfile	READ	An attempt has been made to read a record from a sequential file which is positioned at ENDFILE.
161	WRITE operation failed	WRITE	After repeated retries WRITE(2) could not successfully complete an output operation. This may occur if a signal to be caught interrupts output to a slow device
162	No WRITE	WRITE	An attempt has been made to write to a file which is

	permission		defined for input only.
163	Unit not defined or connected	FIND	The unit specified by a FIND statement is not open. The unit should first be defined by a DEFINE FILE statement, or should be connected by some other means.
164	Invalid channel number	Any I-O Operation	The unit specified in an I/O statement is a negative value.
166	Unit already connected	DEFINE FILE	The unit specified in a DEFINE FILE statement is already open.
167	Unit already defined	DEFINE FILE, OPEN	The same unit has already been specified by a previous DEFINE FILE statement.
168	File already exists	OPEN	An attempt has been made to OPEN an existing file with STATUS=NEW.
169	Output file capacity exceeded	READ, WRITE	An attempt has been made to write to an internal or external file beyond its maximum capacity.
171	Invalid operation on file	Positional, READ, WRITE	An I/O request was not consistent with the file definition; for example, attempting a BACKSPACE on a unit that is connected to the screen.
172	various	READ, WRITE	An unexpected error was returned by READ2 - the error text will be the NT* message associated with the failure.
173	various	READ, WRITE	An unexpected error was returned by WRITE- the error text will be the LINUX* message associated with the failure.
174	various	READ, WRITE	An unexpected error was returned by LSEEK - the error text will be the LINUX message associated with the failure.
175	various	OPEN, CLOSE	An unexpected error was returned by UNLINK - the error text will be the LINUX message associated with the failure.
176	various	OPEN, CLOSE	An unexpected error was returned by CLOSE- the error text will be the LINUX message associated with the failure.
177	various	OPEN	An unexpected error was returned by CREAT - the error text will be the LINUX message associated with the failure.
178	various	OPEN	An unexpected error was returned by OPEN- the error text will be the LINUX message associated with the failure.
181	Substring out of range	Namelist READ	A character substring reference in the input data lay beyond the bounds of the character variable.
182	Invalid variable name	Namelist READ	A name in the data was not a valid variable name.
185	Too many values	Namelist READ specified	A repetition factor (of the form r*c) exceeded the number of elements remaining unassigned in either an array or array element reference.
186	Not enough subscripts	Namelist READ specified	An array element reference contained fewer subscripts than are associated with the array.
187	Too many subscripts	Namelist READ specified	An array element reference contained more subscripts than are associated with the array.

188	Value out of range	Formatted READ	During numeric conversion from character to binary form a value in the input record was outside the range associated with the corresponding I-O item.
190	File not suitable	OPEN	A file which can only support sequential file operations has been opened for direct access I-O.
191	Workspace exhausted	OPEN	Workspace for internal tables has been exhausted.
192	Record too long	READ	The length of the current record is greater than that permitted for the file as defined by the RECL= specifier in the OPEN statement
193	Not connected for unformatted I-O	Unformatted READ/WRITE	An attempt has been made to access a formatted file with an unformatted I-O statement.
194	Not connected for formatted I-O	Formatted READ/WRITE	An attempt has been made to access an unformatted file with a formatted I-O statement.
195	Backspace not permitted	BACKSPACE	An attempt was made to BACKSPACE a file which contains records written by a list-directed output statement; this is prohibited by the ANSI Standard.
199	Field too large	List-Directed READ, Namelist READ	An item in the input stream was found to be more than 1024 characters long (this does not apply to literal constants).
203	POSITION conflict	OPEN	When a file is to be connected to a unit to which it is already connected, then only the BLANK, DELIM, ERR, IOSTAT and PAD specifiers may be redefined. An attempt has been made to redefine the POSITION specifier.
204	ACTION conflict	OPEN	When a file is to be connected to a unit to which it is already connected, then only the BLANK, DELIM, ERR, IOSTAT and PAD specifiers may be redefined. An attempt has been made to redefine the ACTION specifier.
205	No read permission	READ	An attempt has been made to READfrom a unit which was OPENed with ACTION="WRITE".
206	Zero stride invalid	Namelist READ	An array subsection reference cannot have a stride of zero.
208	Incorrect array triplet syntax	Namelist READ	An array subsection triplet has been input incorrectly.
209	Name not a derived type	Namelist READ	A name in the data which is not a derived type has been followed by a ‘%’.
210	Invalid component name	Namelist READ	A derived type reference has not been followed by an ‘=’.
211	Component name expected	Namelist READ	A ‘%’ must be followed by a component name in a derived type reference.
212	Name not in derived type	Namelist READ	A component is not in this derived type.
213	Only one component may be array-valued	Namelist READ	In a derived-type reference, only the derived type or one of its components may be an array or an array subsection.

214	Object not allocated	READ/WRITE	An item has been used which is either an unallocated allocatable array or a pointer which has been disassociated.

Little-Big Endian Conversion Errors

Error	Message	Where Occurring	Description
215	Conversion of derived data types is disabled	READ/WRITE	Conversion of derived data types is disabled if READ/WRITE statement refers to derived data type. (This error is fatal.)
216	!Internal Error! Unknown data size	READ/WRITE	Unknown data size. Fatal error. Contact Intel.
217	!Internal Error! Conversion buffer too small	READ/WRITE	Conversion buffer too small. Fatal error. Contact Intel.

Other Errors Reported by I/O statements

Errors 101-107 arise from faults in run-time formats:

Error	Message
101	Syntax error in format
102	Format is incomplete
103	A positive value is required here
104	Minimum number of digits exceeds width
105	Number of decimal places exceeds width
106	Format integer constants > 32767 are not supported
107	Invalid H edit descriptor

Notes

- The I/O statements OPEN, CLOSE and INQUIRE are classified as Auxiliary I/O statements. The I/O statements REWIND, ENDFILE and BACKSPACE are classified as Positional I/O statements.
- The IOSTAT = variable is set to -1 if an end-of-file condition occurs, to -2 if an end-of-record condition occurs (in a non-advancing READ), to the error number if one of the listed errors occurs, and to 0 if no error occurs.
- Should no input/output specifier relating to the type of the occurring input/output error be given (END=, EOR=, ERR= or IOSTAT=, as appropriate), then the input/output error will terminate the user program. All units which are currently opened will be closed, and the appropriate error message will be output on Standard Error followed (if requested) by a postmortem report (see Runtime Diagnostics).
- The form of an input/output error message is presented in the table below.

I/O Error nnn :	Text of message
In Procedure :	Procedure name
At Line :	Lline number
Statement :	I/O statement type
Unit :	Unit identifier or Internal File
Connected To :	Ffile name

Form :	Formatted, Unformatted or Print
Access :	Sequential or Direct
Nextrec :	Record number
Records Read :	Number of records input
Records Written :	Number of records output
Current I/O Buffer :	Snapshot of the current record with a pointer to the current position

Note

Only as much information as is available or pertinent will be displayed.

Intrinsic Procedure Errors

The following error messages, which are unnumbered, are generated when incorrect arguments are specified to the Intel® Fortran Compiler intrinsic procedures, and option `-CS` was selected at compile-time. The messages are given in alphabetic order.

Each message is preceded by a line of the form:

`ERROR calling the intrinsic subprogram` *`name`*`:`

where *`name`* is the name of the intrinsic procedure called. The term "integer" indicates `integer` format of an argument.

List of Intrinsic Errors

Argument `integer` of the intrinsic function name has string length `integer`. It should have string length at least `integer`.

Argument `integer` of the intrinsic function name is a rank `integer` array. It should be a rank `integer` array.

Argument `integer` of the intrinsic function name is an array with `integer` elements. It should be an array with at least `integer` elements.

Argument *`name`* has the value `integer` and argument *`name`* has the value `integer`. Both arguments should have non-negative values and their sum should be less than or equal to `integer`.

Array argument *`name`* has size `integer`. It should have size int `integer` .

Array arguments *`name1`* and *`name2`* should have the same shape.
The shape of argument *`name1`* is: (`integer`, `integer`,..., `integer`).
The shape of argument *`name2`* is: (`integer`, `integer`,..., `integer`).

At least one of the array arguments should have `rank = 2`
The extent of the last dimension of `MATRIX_A` is `integer`. The extent of the first dimension of `MATRIX_B` is `integer`. These values should be equal.

The `DIM` parameter had a value of `integer`. Its value should be `integer`.

The `DIM` parameter had a value of `integer`. Its value should be at least integer and no larger than `integer`.

The *`name`* array has shape: (`integer`, `integer`,..., `integer`).
The shape of *`name`* should be: (`integer`, `integer`,..., `integer`).

The NCOPIES argument has a value of integer. NCOPIES should be non-negative.
The ORDER argument should be a permutation of the integer1 to integer.
The contents of the ORDER argument array is: (integer, integer,..., integer).
The rank of the RESULT array should be equal to the size of the SHAPE array.
The rank of the RESULT array is integer. The size of the integer array is integer.
The RESULT array has shape: (integer, integer,..., integer).
The shape of the RESULT array should be: (integer, integer,..., integer).
The RESULT array has size integer. It should have size integer.
The RESULT character string has length integer. It should have length integer.
The SHAPE argument has size integer.
Its size should be at least integer and no larger than integer.

- The SHAPE argument should have only non-negative elements.
- The contents of the SHAPE array is: (integer, integer,..., integer).
- The SIZE argument has a value integer. Its value should be non-negative.
- The size of the SOURCE array should be at least integer.
- The size of the SOURCE array is integer.
- When setting seeds with the intrinsic function name, the first seed must be at least integer and not more than integer, and the second seed must be at least integer and not more than integer.

Mathematical Errors

This section lists the errors that can be reported as a consequence of using an intrinsic function or the exponentiation operator **.

If any of the errors below is reported, the user program will terminate. A postmortem report (see Runtime Diagnostics) will be output if the program was compiled with the option -d{n}. All input-output units which are open will be closed.

The number and text of mathematical errors are:

Error	Message
16	Negative DOUBLE PRECISION value raised to a non-integer power
17	DOUBLE PRECISION zero raised to non-positive power
22	REAL zero raised to non-positive power
23	Negative REAL value raised to a non-integer power
24	REAL value raised to too large a REAL power
38	INTEGER raised to negative INTEGER power
39	INTEGER zero raised to non-positive power
40	INTEGER to INTEGER power overflows
46	DOUBLE PRECISION value raised to too large a DOUBLE PRECISION power
47	COMPLEX zero raised to non-positive INTEGER power

Exception Messages

The following messages, which are unnumbered, are a selection of those which can be generated by exceptions (signals). They indicate that a hardware-detected or an asynchronous error has occurred. Note that you can obtain a postmortem report when an exception occurs by compiling with the -d{n} option.

The occurrence of an exception usually indicates that the Fortran program is faulty.

Message	Comment
`**QUIT signal**`	Program aborted by the user typing ^/ (ctrl + /)
`**Illegal Instruction**`	May be indicative of a bad call on a function that is defined to return a derived type result: either the sizes of the expected and actual results do not correspond, or the function has not been called as a derived type function.
`**Alignment Error**`	Access was attempted to a variable which is not aligned on an address boundary appropriate to its type; this could occur, for example, when a formal double-precision type variable is aligned on a single word boundary.
`**Address Error** **Bus Error**`	Usually caused by a wrong value being used as an address (check the associativity of all pointers).